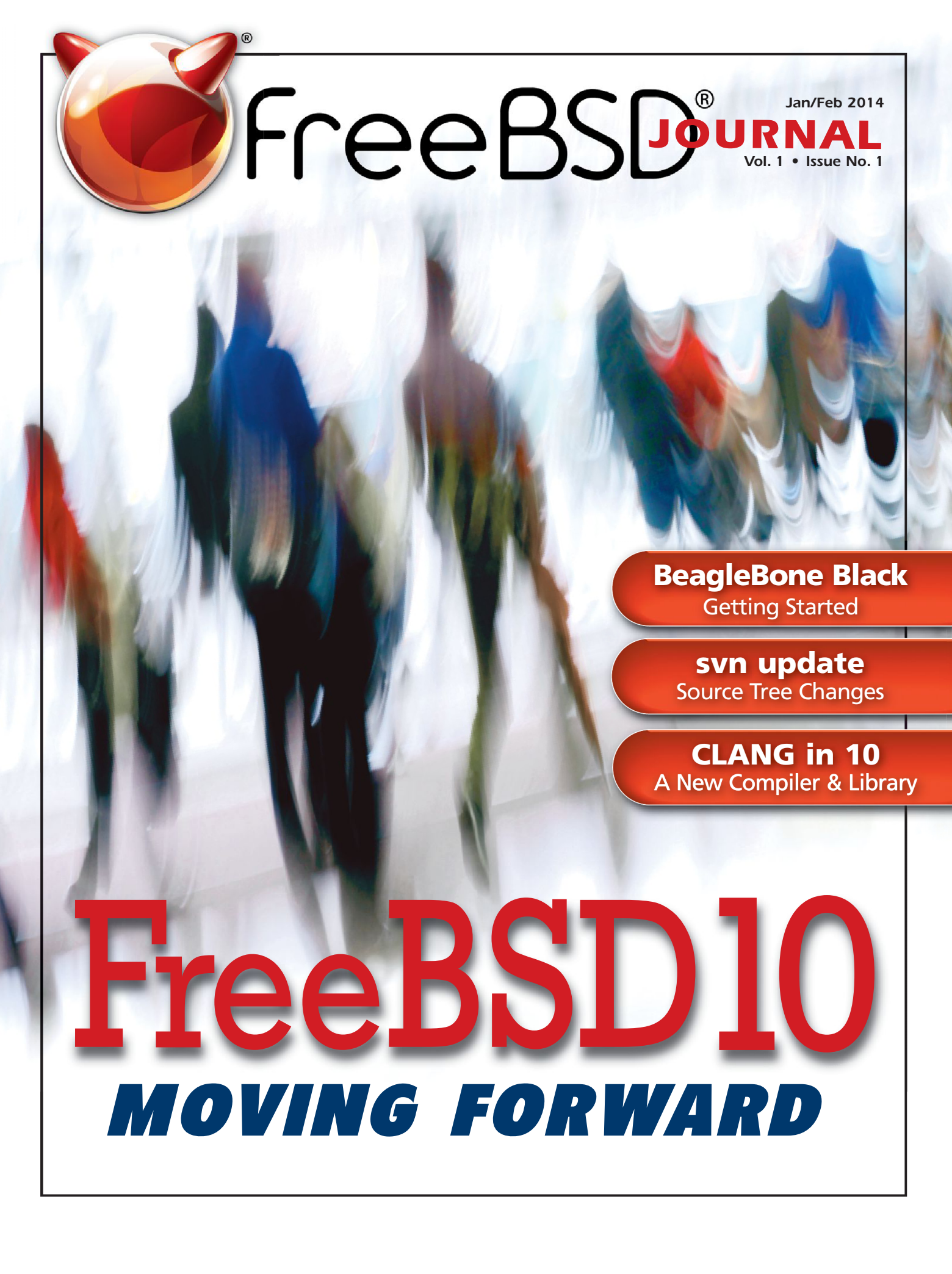


FreeBSD® JOURNAL
Jan/Feb 2014
Vol. 1 • Issue No. 1

BeagleBone Black
Getting Started

svn update
Source Tree Changes

CLANG in 10
A New Compiler & Library

FreeBSD 10

MOVING FORWARD

Table of Contents • Jan/Feb 2014

FreeBSD Journal • Vol. 1, Issue No. 1

DEPARTMENTS & COLUMNS

FEATURE ARTICLES

FreeBSD® Journal

Editorial Board

John Baldwin • Member of the FreeBSD Core Team

Daichi Goto • Director at BSD Consulting Inc. (Tokyo)

Joseph Kong • Author of *FreeBSD Device Drivers*

Dru Lavigne • Director of the FreeBSD Foundation and Chair of the BSD Certification Group

Michael W. Lucas • Author of *Absolute FreeBSD*

Kirk McKusick • Director of the FreeBSD Foundation and lead author of *The Design and Implementation* book series

George Neville-Neil • Director of the FreeBSD Foundation, and co-author of *The Design and Implementation of the FreeBSD Operating System.*

Hiroki Sato • Director of the FreeBSD Foundation, Chair of AsiaBSDCon, member of the FreeBSD Core Team, and Assistant Professor at Tokyo Institute of Technology.

Robert Watson • Director of the FreeBSD Foundation and founder of the TrustedBSD Project. Lecturer at the University of Cambridge

S&W Publishing LLC
PO BOX 408, BELFAST, MAINE 04915

Publisher • Walter Andrzejewski
walter@FreeBSDjournal.com

Editor-at-Large • James Maurer
jmaurer@FreeBSDjournal.com

Copy Editor • Annaliese Jakimides
annaliese@FreeBSDjournal.com

Art Director • Dianne M. Kischitz
dianne@FreeBSDjournal.com

Office Administrator • Cindy DeBeck
cindy@FreeBSDjournal.com

Advertising Sales • Walter Andrzejewski
walter@FreeBSDjournal.com
Call 888/290-9469

FreeBSD Journal (ISBN: 978-0-615-88479-0) is published 6 times a year. (January/February, March/April, May/June, July/August, September/October, November/December).

Published by the FreeBSD Foundation,
PO Box 20247, Boulder, CO 80308
ph: 1/720-207-5142 • fax: 1/720-222-2350
email: board@freebsdfoundation.org

LETTER
from the Board

WELCOME
To the Entire FreeBSD Community!

Welcome to the premiere issue of *FreeBSD Journal*, our brand new, professionally-produced, online magazine available as a subscription or single issues through the various app stores, including Apple iTunes, Google Play, and Amazon Kindle.

In upcoming issues you will find feature-length articles and columns that address the entirety of the FreeBSD community. Our editorial coverage will be diverse, including topics like: managing large-scale system deployments, application development, systems programming, embedded systems, academic research, and general software development. All using FreeBSD.

This issue (Issue No.1) is dedicated to FreeBSD 10, the latest in the FreeBSD Project's line of major releases. The release of FreeBSD 10 brings many new features not seen in other open-source operating systems, including a brand new compiler toolchain based on LLVM, as well as mature support for ZFS as a first-class, kernel-based filesystem.

FreeBSD Journal will publish six issues per year. The Editorial Board has already planned the first year with issues that will cover FreeBSD 10, pkgng (Packaging Next Generation), Networking, Virtualization, Development Tools, and Support for New Hardware Features.

- FreeBSD 10
- pkgng (Packaging Next Generation
- Networking
- Virtualization
- Development Tools
- Support for New Hardware Features

FreeBSD Journal is supported by the FreeBSD Foundation (www.freebsdfoundation.org), a 501(c)(3), not-for-profit organization, whose sole purpose is to help the FreeBSD Project grow and flourish. Your subscriptions and the advertising revenue the journal produces help to offset the costs of acquiring, developing, designing, and delivering this information to you.

We trust you'll like what you see in *FreeBSD Journal* and hope that you'll tell everyone you know about it. Thanks for taking a look.

Sincerely,

FreeBSD Journal
Editorial Board

BY DAVID CHISNALL

Clang in 10

FreeBSD 10 includes out-of-the-box support for the majority of the C11 and C++11 standards.

The road to get us to

this point has been long and has involved a lot of replacements of large parts of the toolchain. We're now shipping clang as the default C/C++ (and Objective-C) compiler and libc++ as the default C++ standard library implementation.

So why a new compiler and a new C++ standard library? The GNU Compiler Collection has been a part of FreeBSD for most of its life. There were painful upgrades from 2.x to 3.x and then to 4.x. It was always a somewhat sore spot for the project that a BSD-licensed operating system depended on a GPL'd compiler.

Some History

In 2007, GCC went to GPLv3. This license had one or two clauses that some major downstream consumers found unacceptable and so the decision was made not to import any GPLv3 code into the base system. The version of GCC stayed at 4.2.1.

In 2008, some developers at Apple released clang, a C front end for the LLVM compiler infrastructure. LLVM was originally offered to the Free Software Foundation as a new back end for GCC, but was turned down. Apple started using the LLVM back end (and hired the original developer of LLVM and a lot of other people) and continued to improve it.

LLVM is far more than just a C compiler. It provides a uniform intermediate representation that language front ends can generate, optimisers can modify, and back ends can convert into native code. One of its earliest was in Apple's OpenGL shader stack, where a naive LLVM-based JIT compiler outperformed the handwritten one by around 20% and worked

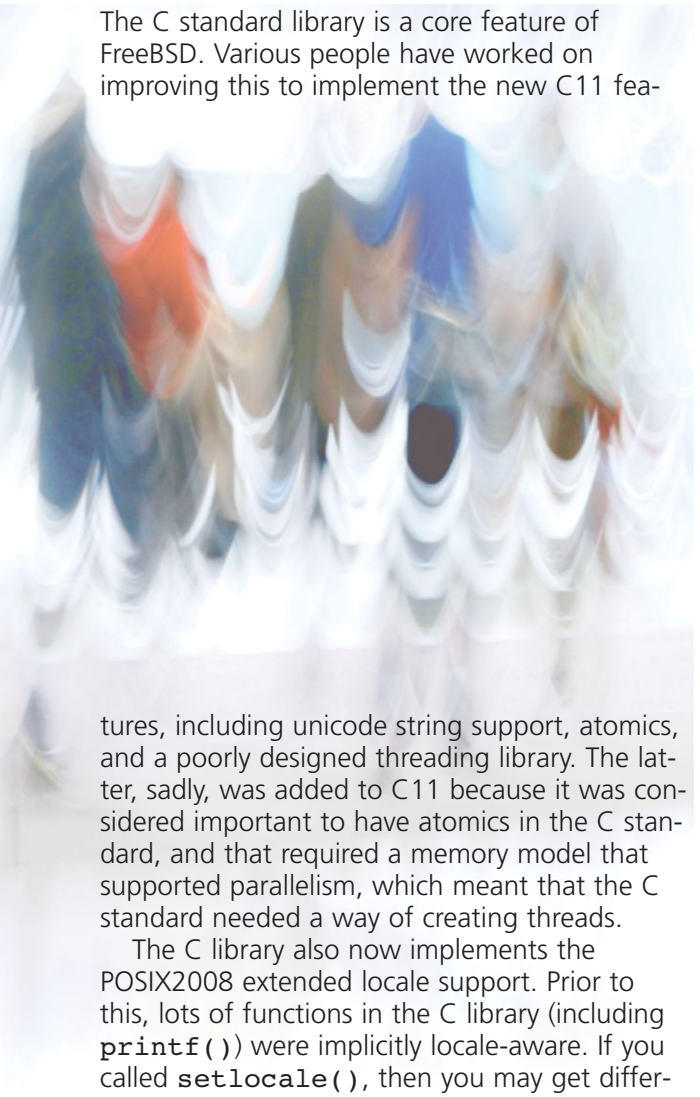
on all of Apple's supported architectures.

In the last five years, the combination of clang and LLVM has become a mature product. It's now the only compiler supported by Apple and is one of the standard compilers shipped with the Android NDK. Companies like ARM, Qualcomm, Apple, Google, AMD, Intel, and many others are contributing large amounts of code to it.

Meanwhile, our old GCC has begun to look quite dated. At the end of 2011, the C and C++ standards committees released specifications for new dialects of their respective languages. The extensions to C were relatively simple. The changes to C++ were huge. Both required changes to both the compiler and the standard library.

The C Standard Library

The C standard library is a core feature of FreeBSD. Various people have worked on improving this to implement the new C11 features, including unicode string support, atomics, and a poorly designed threading library. The latter, sadly, was added to C11 because it was considered important to have atomics in the C standard, and that required a memory model that supported parallelism, which meant that the C standard needed a way of creating threads.



The C library also now implements the POSIX2008 extended locale support. Prior to this, lots of functions in the C library (including `printf()`) were implicitly locale-aware. If you called `setlocale()`, then you may get different results from them. This includes fairly small things, like the separator character for floating point values. The `setlocale()` function sets the locale globally, which means that it's not safe for multiple locales in a multithreaded program.

The POSIX2008 locale extensions provide a per-thread locale, but more importantly they provide a set of variants of the standard C functions that take an explicit locale as a parameter. Lots of things use these APIs, including new versions of GNOME, but the primary consumer that we're interested in here is libc++.

Libc++ is another component of the stack to originate at Apple. Mostly developed by Howard Hinnant, it is a completely new implementation of the C++ standard library, designed from scratch for C++11. Implementing C++11 support in the standard library required quite invasive changes (which broke backwards compatibility) and so seemed like a good place to start from scratch. This also allowed all of the standard data structures to be redesigned in a way that makes more sense for modern hardware, for example focusing more on cache usage in std::string.

The Benefits of C++11 and C11

Now that we've got a new C++ stack and an improved C stack, what does that give us? I briefly mentioned some of the new features in C11, but my personal favorite is the new atomic types. These are declared with the `_Atomic()` type qualifier, for example:

```
_Atomic(int) x;
```

The qualifier ensures that you don't accidentally mix atomic and non-atomic accesses to the same variable. Simple operations will automatically become atomic. Some care is needed here. For example, consider the following three lines:

```
x += 1;
x++;
x = x + 1;
```

The first two will be a sequentially-consistent atomic increment. The last, however, will be a sequentially-consistent atomic read, an add, and then an atomic store. Sequentially consistent means that atomic operations are all globally visible in the same order. For example, if one core increments x and another increments y, then either the increment to x or y is visible first, but it is the same on all cores.

At the opposite extreme, atomic operations with relaxed ordering require that the operation be atomic with respect to that single variable, but not with respect to others. If x and y were relaxed, then it would be acceptable for some threads to see the new x and the old y, and some to see the new y and old x.

The stdatomic.h header contains a lot of functions that operate on atomic variables, and our implementation contains several code paths to make it work with old compilers. This is a pattern that we've replicated elsewhere and things like the `_Thread_local` storage qualifier and similar are implemented in our standard headers using extensions when using a compiler that does not support them natively.

One other addition in C11 has made it possible to clean up some of our headers. The standard adds `_Generic()` expressions, which are similar to switch statements selecting based on the type, rather than the value, of an expression. This is only useful in macros, but it's useful in several standard macros that must be defined in C header files. In particular, there are several related to numerics that are defined for all floating-point types.

Two examples in this category are `isinf()` and `isnan()`, which return true if the argument is infinite or a not-a-number value, respectively. Our old code was determining the correct path to call depending on the size of the argument. This meant that if you passed a 32-bit `int` value, it would call the function that expected a 32-bit `float`. This would always return false (because no `float` that is created by casting an int can possibly be infinite or not-a-number), but almost certainly hid a logic bug because there's no reason why you'd ever want to check these properties on an int.

We now use `_Generic()` for these and so they will always go to the correct function and you get an error if you try to call it with the wrong argument. We found a few bug ports as a result, and some quite bizarre behavior. For example, both Google's v8 and Mono had configure script checks that tested whether `isnan(1)` worked. In the Mono case, if they detected that `isnan(int)` didn't work, then they declared their own `isnan(double)` to use, which then conflicted with the system one.

Challenges with Clang

Getting the system ready for clang, initially as the system compiler and then as the only compiler in the base system, has been an interesting experience. A lot of the initial experiences were simply from getting FreeBSD to compile without warnings. Clang gives a lot more warnings than our old gcc (gcc 4.8 is now at a similar quality, although still has a slight tendency toward false positives) and we try to ensure that all of our code builds without warnings. Somewhat amusingly, the worst offender in our tree for having compiler warnings was gcc itself.

In the ports tree, things were somewhat different. We do maintain local patches for a lot of programs, but ideally these should be small (and should be pushed upstream where possible).

Most C code works fine with clang. The biggest issue that we faced in the ports tree was that clang defaults to C99 as the dialect when invoked as `clang` or `cc`, whereas gcc defaults to C89. It's somewhat depressing that people still invoke a C compiler as cc in 2013, because cc was deprecated in the 1997 release of POSIX and defined as accepting an unspecified version of the C language. Back then, the choices were K&R C or C89. If you wanted C89, you were recommended to invoke the `c89` utility. The next release of POSIX added a `c99` utility. Presumably the next one to be published will also specify a `c11` utility.

The C99 specification was carefully designed so that valid C89 programs were also valid C99 programs, so this shouldn't have been a problem. Unfortunately, this didn't quite work because few people wrote C89 code, instead they wrote C89 with GNU extensions. I said GCC defaulted to C89 mode, but that's not quite correct: it defaulted to C89 with GNU extensions (`gnu89`, as it's known on the command line).

There is only one significant incompatibility between C89 with GNU extensions and C99, and that's the handling of the `inline` keyword. The differences meant that code that expected the GNU inline rules would end up with functions defined multiple times in C99 mode and so would fail to link. This is relatively easy to fix—just add `-fgnu89-inline` to the compiler flags—but it needed to be done for every port that had this kind of error. When you have over 20,000 ports, even simple fixes are a lot of work.

In C++, the problems were more pronounced. The rules for symbol resolution in C++ are incredibly complicated. This is especially true inside templates, where the standard calls for a two-stage lookup. Both GCC and the Microsoft C++ compiler managed to get this wrong. Of course, they did it in different wrong ways,

which was why it has traditionally been very difficult to move C++ code between compilers.

Clang benefited from all of this experience and wrote the C++ parser to the letter of the specification. This means that any standards-compliant C++98 code will compile with clang. These days, so does C++11 code, and some C++1y code (C++1y is the draft that will most likely become C++14). Unfortunately, when you refuse to compile a popular open source program, you don't get much sympathy when you turn around and say—"well, the code is invalid."

The Challenge of Migration

For C code, there is no difficulty migrating. The C ABI is defined by the target platform and both Clang and GCC generate entirely compatible code. This is also true for C++ code, if you're only talking about C++—the language. Unfortunately, there is more to either language—there is also the standard library. In the case of C, this is FreeBSD libc. Again, this is shared between compilers.

In the case of C++, it's actually two libraries. The smaller of the two implements the dynamic parts of the language such as exceptions and the `dynamic_cast<>` operator. The larger implements the standard template library (STL). In our old stack, these were implemented by libsupc++ and libstdc++, respectively. Originally, these two were statically linked together.

In the new stack, these are libcxxrt and libc++. As part of the migration path, we wanted to make it possible for programs to link against libraries that used both libc++ and libstdc++. This required modifying our libstdc++ to link against libsup++ as a dynamic (shared) library, which then allowed libmap.conf to switch between them.

Unfortunately, life is never that simple. ELF symbol versioning associates the symbol with both the version and the library that it came from and so existing binaries would fail to link on symbols suddenly moved from `libstdc++.so` to `libsupc++.so`. The solution to this was to make `libstdc++` a filter library. This allows it to, effectively, forward the symbol resolution on to libraries it linked against.

With this done, it became possible to link against both. The STL symbols have different names and so you will use the ones from whichever headers you included in the source code. Unfortunately, because they have different symbol names (and different binary layouts), you can't use them interchangeably. If you have a library that has interfaces that use STL types (for example, `std::string`) then both the library and the things that call it must use the same STL implementation.

This causes some problems in the ports collection, because a few libraries won't compile with clang and so can't use libc++, whereas others require C++11 and so won't compile with our base GCC. Using a GCC from ports doesn't really address this either, as many of the old C++ programs also won't compile with a new GCC, and the new libstdc++ is not binary-compatible with the one that we have in the base system either.

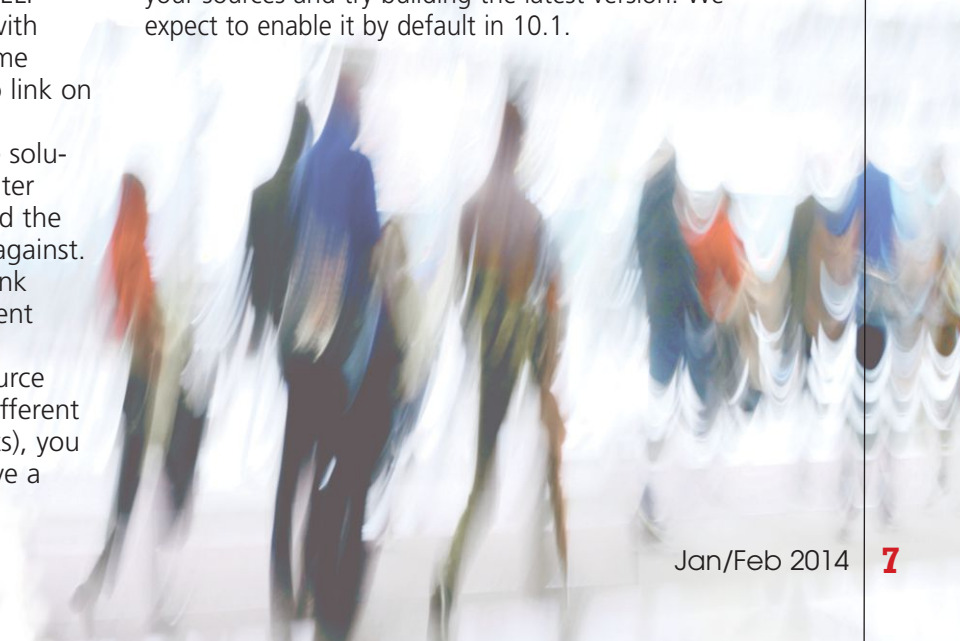
Debugging

One other unfortunate problem with the clang switch is that clang now emits debug information conforming to version 4 of the DWARF standard. Soon, it will default to DWARF 5, which includes support for much smaller debug info tables and for separating out the debug info into separate files during compilation so that they can be linked separately.

Unfortunately, the old version of the GNU debugger (GDB) that we include in the base system can only support DWARF 2. For 10, we've imported the LLVM Debugger (LLDB) and Ed Maste has been working (with FreeBSD Foundation funding) on the FreeBSD port.

LLDB, like the rest of LLVM, is very modular. It is intended as a set of libraries that allow debugging features to be added to various applications, ranging from command-line tools to IDEs. It is largely developed by Apple and so remote debugging was a core part of the design, allowing ARM devices to be debugged from x86 desktops.

All of these are nice features, but unfortunately LLDB isn't quite ready for enabling by default in the 10 release. It's in the tree, so feel free to upgrade your sources and try building the latest version. We expect to enable it by default in 10.1.



Clang in 10

Architectural Problems

These days, FreeBSD has one tier 1 architecture: x86 (in 32-bit and 64-bit variants). ARMv6 and newer are very close to being tier 1 as well. These two are well supported by Clang and by the LLVM back end. Unfortunately, we also have a lot of tier 2 architectures, such as SPARC, PowerPC, and MIPS, with less good support.

LLVM has quite good support for 64-bit PowerPC, developed largely by Argone National Laboratory, but not nearly as good support for 32-bit PowerPC. Since Apple switched to Intel, these architectures have been dead in the consumer PC market, but they're still popular in a lot of places where people ship embedded FreeBSD-derived systems, especially in the automotive industry.

The MIPS back end is now able to compile LLVM itself, which is quite an achievement given the size and complexity of the code base, but it's still a little way away from being able to compile the FreeBSD kernel.

SPARC and IA64 are two with less certain futures. The SPARC back end in LLVM is one of the oldest ones, yet it is still not production-ready. The Itanium back end was removed, after being unmaintained for a while. Intel doesn't seem to be pushing Itanium very hard, and Oracle seems to regard SPARC as a platform for running Solaris-based Oracle appliances, so the future of these architectures is not that certain anyway, but it would be a shame to drop support for them in FreeBSD while there are still systems using them.

Going Forward

The goal in all of this was to make FreeBSD a modern development platform. We've achieved that. FreeBSD 10 shipped with the most complete C11 and C++11 (and C++1y) implementations of any system to date. We now have a modern compiler and C++ stack, with an active upstream community that is engaged with FreeBSD as a consumer, and a number of people (myself included) who contribute to both projects.

We still have a few missing pieces for a completely BSD-licensed toolchain, however. We currently ship a lot of GNU binutils. Some things, such as the GNU assembler, are easy to replace. The LLVM libraries contain all of the required functionality; they just require small tools to be written to implement them.

The one exception is the linker. Like compilers, linkers are quite complex pieces of software. We're currently evaluating two linkers to replace GNU ld. The first, MCLinker, was originally developed by MediaTek using LLVM libraries and now has a larger community. It currently ships, was one of the linkers in the Android SDK, and can link all of the base system, but lacks support for symbol versioning (this may have been finished by the time you're reading this, as work is ongoing to implement it).

The other option is lld, the LLVM linker. This is a more complex design and is not yet as advanced, but does have some large corporate backers such as Sony (Sony is a FreeBSD consumer), and so might be a better long-term prospect.

Whichever we select, FreeBSD will continue to pick the best tools for the job. We hope to have a fully BSD licensed toolchain by default for 11.0, and as optional components in the 10.x series. Being BSD licensed is always nice, but we won't switch until the tools are also better. ●

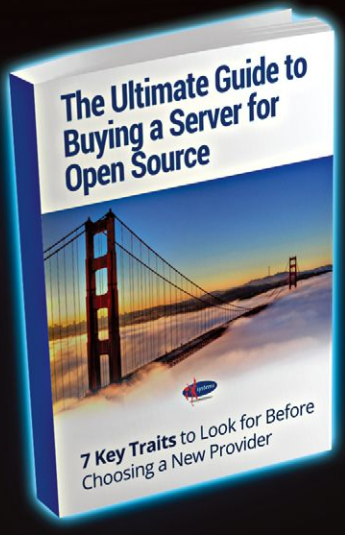
David Chisnall is a researcher at the University of Cambridge Computer Laboratory and a member of the FreeBSD Core Team. He is also an active contributor to several other open source projects, including LLVM, GNUstep, and Étoilé. In between writing code, he has written several books. When not in front of a computer, he dances Cuban salsa and Argentine tango.

Certified FreeBSD Servers...*Actually* Built By Champions Of FreeBSD

"iXsystems Takes The Headache Out Of Server Shopping."

Allan Jude - *www.scaleengine.com*

Savvy FreeBSD pros like Allan Jude know there is a lot more to choosing a server provider than finding the cheapest quote. If your company is expanding, starting a new project, or looking to upgrade hardware then check out these tips to avoid some of the common pitfalls of buying new hardware.



What To Look For

1. Free Pre-Purchase Consultation

The test of a good server provider starts before the first sales proposal arrives in your inbox. You'll know they want more than a sale when they spend more time figuring out what you actually need instead of forcing a cookie-cutter solution.

2. Servers Specifically Built For FreeBSD

The big name vendors are not experts in FreeBSD and that is a FACT. Instead, look for companies that actually have FreeBSD experts in-house. This will ensure you get a server that is finely tuned for FreeBSD.

To learn about the 7 key traits you must absolutely demand from your server provider, go to iXsystems.com/FreeBSDJournal/ and grab your free copy of **The Ultimate Guide To Buying A Server For Open Source.**

Featuring Intel® Xeon® Processors

Call iXsystems toll free today! **1-855-GREP-4-IX**

BY JOHN BALDWIN

IMPLEMENTING SYSTEM CONTROL NODES (sysctl)

The FreeBSD kernel provides a set of system control nodes that can be used to query and set state information. These nodes can be used to obtain a wide variety of statistics and configure parameters. The information accepted and provided by each node can use one of several types-including integers, strings, and structures.

THE NODES are organized as a tree. Each node is assigned a unique number within its level. Nodes are identified internally by an array of node numbers that traverses the tree from the root down to the node being requested. Most nodes in the tree have a name, and named nodes may be identified by a string of names separated by dots. For example, there is a top-level node named "kern" that is assigned the number one (1), and it contains a child node named "ostype" that is assigned the number one (1). This node's address is 1.1, but it may also be referred to by its full name: "kern.ostype". Users and applications generally use node names rather than node numbers.

Accessing System Control Nodes

The standard C library on FreeBSD provides several routines for accessing system control nodes. The sysctl(3) and sysctlbyname(3) functions access a single node. The sysctl(3) function uses the internal array of numbers to identify a node while sysctlbyname(3) accepts a string containing the node's full name. Each sysctl access can

query the current state of the node, set the state of the node, or perform both operations.

The sysctlnametomib(3) function maps a node's full name to its internal address. This operation uses an internal sysctl node and is a bit expensive, so a program that queries a control node frequently can use this routine to cache the address of a node. It can then query the node using sysctl(3) rather than sysctlbyname(3).

Some control nodes have a named prefix with unnamed leaves. An example of this is the "kern.proc.pid" node. It contains a child node for each process. The internal address of a given process's node consists of the address of "kern.proc.pid" and a fourth number which corresponds to the pid of the process.

Example 1 demonstrates using this to fetch information about the current process.

```
struct kinfo_proc kp;
int i, mib[4];
size_t len;

/* Fetch the address of the "kern.proc.pid"
   prefix. */
len = 3;
sysctlnametomib("kern.proc.pid", mib, &len);

/* Fetch the process information for the
   current process. */
len = sizeof(kp);
mib[3] = getpid();
sysctl(mib, 4, &kp, &len, NULL, 0);
```

Example 1

Simple Control Nodes

In the kernel, the <sys/sysctl.h> header provides several macros to declare control nodes. Each declaration includes the name of the parent node, a number to assign to this node, a name for the node, flags to control the node's behavior, and a description of the node. Some declarations require additional arguments. The parent node is identified by its full name, but with a single underscore as a prefix and dots replaced by underscores. For example, the "foo.bar" parent node would be identified by "_foo_bar". To declare a top-level node, use an empty parent name. The number should use the macro `OID_AUTO` to request that the system assign a unique number. (Some nodes use hardcoded numbers for legacy reasons, but all new nodes should use system-assigned numbers.) The flags argument must indicate which types of access the node supports (read, write, or both) and can also include other optional flags. The description should be a short string describing the node. It is displayed instead of the value when the "-d" flag is passed to sysctl(8).

Integer Nodes

The simplest and most common control node is a leaf node that controls a single integer. This type of node is defined by the `SYSCTL_INT()` macro. It accepts two additional arguments; a pointer and a value. If the pointer is non-`NULL`, it should point to an integer variable which will be read and written by the control node (as specified in the flags argument). If the pointer is `NULL`, then the node must be a read-only node that returns the value argument when read.

```
SYSCTL_INT(_kern, OID_AUTO, one, CTLFLAG_RD,
    NULL, 1, "Always returns one");

int frob = 500;
SYSCTL_INT(_kern, OID_AUTO, frob, CTLFLAG_RW,
    &frob, 0, "The \"frob\" variable");
```

Example 2

Example 2 defines two integer sysctl nodes: "kern.one" is a read-only node that always returns the value one and "kern.frob" is a read-write node that reads and writes the value of the global "frob" integer.

Additional macros are available for several integer types including: `SYSCTL_UINT()` for unsigned integers, `SYSCTL_LONG()` for signed long integers, `SYSCTL_ULONG()` for unsigned long integers, `SYSCTL_QUAD()` for signed 64-bit integers, `SYSCTL_UQUAD()` for 64-bit unsigned integers, and `SYSCTL_COUNTER_U64()` for 64-bit unsigned integers managed by the counter(9) API. Only the `SYSCTL_INT()` and `SYSCTL_UINT()` macros may be used with a `NULL` pointer. The other macros require a non-`NULL` pointer and ignore the value parameter.

Other Node Types

The `SYSCTL_STRING()` macro is used to define a leaf node with a string value. This macro accepts two additional arguments: a pointer and a length. The pointer should point to the start of the string. If the length is zero, the string is assumed to be a constant string and attempts to write to the string will fail (even if the node allows write access). If the length is non-zero, it specifies the maximum length of the string buffer (including a terminating null char-

```
static SYSCTL_NODE(, OID_AUTO, demo, 0, NULL,
    "Demonstration tree");

static char name_buffer[64] = "initial name";
SYSCTL_STRING(_demo, OID_AUTO, name,
    CTLFLAG_RW, name_buffer,
    sizeof(name_buffer), "Demo name");

static struct demo_stats {
        int demo_reads;
        int demo_writes;
} stats;
SYSCTL_STRUCT(_demo, OID_AUTO, status,
     CTLFLAG_RW, &stats, demo_stats,
     "Demo statistics");

SYSCTL_OPAQUE(_demo, OID_AUTO, mi_switch,
    CTLFLAG_RD, &mi_switch, 64, "Code",
    "First 64 bytes of mi_switch()");
```

Example 3

acter) and attempts to write a string longer than the buffer's size will fail.

The `SYSCTL_STRUCT()` macro is used to define a leaf node whose value is a single C struct. This macro accepts one additional pointer argument which should point to the structure to be controlled. The size of the structure is inferred from the type.

The `SYSCTL_OPAQUE()` macro is used to define a leaf node whose value is a data buffer of unspecified type. The macro accepts three additional arguments: a pointer to the start of the data buffer, the length of the data buffer, and a string describing the format of the data buffer.

The `SYSCTL_NODE()` macro is used to define a branch node. This macro accepts one additional argument which is a pointer to a function handler. For a branch node with explicit leaf nodes (declared by other `SYSCTL_*()` macros) the pointer should be `NULL`. The macro may be prefixed with `static` to declare a branch node private to the current file. A public node can be forward declared in a header for use by other files via the `SYSCTL_DECL()` macro. This macro accepts a single argument which is the full name of the node specified in the format used for a parent node in the other macro invocations. Example 3 defines a top level node with three leaf nodes describing a string buffer, a structure, and an opaque data buffer.

Node Flags

Each node definition requires a flags argument. All leaf nodes and branch nodes with a non-NULL function handler must specify the permitted access (read and/or write) in the flags field. The flags field can also include zero or more of the flags listed in Table 1.

Complex Control Nodes

System control nodes are not limited to simply reading and writing existing variables. Each leaf node includes a pointer to a handler function that is invoked when the node is accessed. This function is responsible for returning the "old" value of a node as well as accepting "new" values assigned to a node. The standard node macros such as `SYSCTL_INT()` use predefined handlers in `sys/kern/kern_sysctl.c`.

A leaf node with a custom handler function is defined via the `SYSCTL_PROC()` macro. In addition to the standard arguments accepted by the other macros, `SYSCTL_PROC()` accepts a

Table 1

FLAG	PURPOSE
CTLFLAG_ANYBODY	All users can write to this node. Normally only the superuser can write to a node.
CTLFLAG_SECURE	Can only be written if securelevel is less than or equal to zero.
CTLFLAG_PRISON	Can be written to by a superuser inside of a prison created by jail(2).
CTLFLAG_SKIP	Hides this node from iterative walks of the tree such as when sysctl(8) lists nodes.
CTLFLAG_MPSAFE	Handler routine does not require Giant. All of the simple node types set this flag already. It is only required explicitly for nodes that use a custom handler.
CTLFLAG_VNET	Can be written to by a superuser inside of a prison if that prison contains its own virtual network stack.

FLAG	MEANING
CTLTYPE_NODE	This node is a branch node and does not have an associated value.
CTLTYPE_INT	This node describes one or more signed integers.
CTLTYPE_UINT	This node describes one or more unsigned integers.
CTLTYPE_LONG	This node describes one or more signed long integers.
CTLTYPE_ULONG	This node describes one or more unsigned long integers.
CTLTYPE_S64	This node describes one or more signed 64-bit integers.
CTLTYPE_U64	This node describes one or more unsigned 64-bit integers.
CTLTYPE_STRING	This node describes one or more strings.
CTLTYPE_OPAQUE	This node describes an arbitrary data buffer.
CTLTYPE_STRUCT	This node describes one or more data structures.

Table 2

pointer argument named "arg1", an integer argument named "arg2", a pointer to the handler function, and a string describing the format of the node's value. The flags argument is also required to specify the type of the node's value.

Node Types

The type of a node's value is specified in a field in the node's flags. The standard node macros all work with a specific type and adjust the flags argument to include the appropriate type. The `SYSCTL_PROC()` macro does not imply a specific type, so the type must be specified explicitly. Note that all nodes are allowed to return or accept an array of values and the type simply specifies the type of one array member. The standard node macros all return or accept a single value rather than an array. The available types are listed in Table 2.

Note that since `SYSCTL_PROC()` only defines leaf nodes, `CTLTYPE_NODE` should not be used. Branch nodes with custom handlers are described below.

Node Format Strings

Each node has a format string in addition to a type. The sysctl(8) utility uses this string to format the node's value. As with the node type, most of the standard macros specify the format implicitly. The `SYSCTL_OPAQUE` and `SYSCTL_PROC` macros require the format to be specified explicitly. Most format strings are tied to a specific type and most types only have a single format string. The available format strings are listed in Table 3.

Handler Functions

A system control node handler can be used to provide additional behavior beyond reading and writing an existing variable. Handlers can be used to provide input validation such as range checks on new node values. Handlers can also generate temporary data structures to return to userland. This is commonly done for handlers which return a snapshot of system state such as a list of open network connections or the process table.

Handler functions accept four arguments and return an integer error code. The `<sys/sysctl.h>` header provides a macro to define the function arguments: `SYSCTL_HANDLER_ARGS`. It defines four arguments: "oidp", "arg1", "arg2", and "req". The

FORMAT	MEANING
"A"	An ASCII string. Used with CTLTYPE_STRING.
"I"	A signed integer. Used with CTLTYPE_INT.
"IU"	An unsigned integer. Used with CTLTYPE_UINT.
"IK"	An integer whose value is in units of one-tenth of a degree Kelvin. The sysctl(8) utility will convert the value to Celsius before displaying. Used with CTLTYPE_UINT.
"L"	A signed long integer. Used with CTLTYPE_LONG.
"LU"	An unsigned long integer. Used with CTLTYPE_ULONG.
"Q"	A signed 64-bit integer. Used with CTLTYPE_S64.
"QU"	An unsigned 64-bit integer. Used with CTLTYPE_U64.
"S,<foo>"	A C structure of type `struct foo`. Used with CTLTYPE_STRUCT. The sysctl(8) utility understands a few structure types such as `struct timeval` and `struct loadavg`.

Table 3

"oidp" argument points to the `struct sysctl_oid` structure that describes the node whose handler is being invoked. The "arg1" and "arg2" arguments hold the values assigned to the "arg1" and "arg2" arguments to the `SYSCTL_PROC()` invocation that defined this node. The "req" argument points to a `struct sysctl_req` structure that describes the specific request being made. The return value should be zero on success or an error number from `<sys/errno.h>` on failure. If `EAGAIN` is returned, then the request will be retried within the kernel without returning to userland or checking for signals.

Example 4 defines two integer nodes with a custom handler that rejects attempts to set an invalid value. It uses the predefined handler function `sysctl_handle_int()` that is used to implement `SYSCTL_INT()` to update a local variable. If the request attempts to set a new value, it validates the new value and only updates the associated variable if the new value is accepted.

This example uses a predefined handler (`sysctl_handle_int()`) to publish the old value and accept a new value. Some custom handlers need to manage these steps directly. The macros `SYSCTL_IN()` and `SYSCTL_OUT()` are provided for this purpose. Both macros accept three arguments: a pointer to the current request ("req") from `SYSCTL_HANDLER_ARGS`, a pointer to a buffer in the kernel's address space, and a length. The `SYSCTL_IN()` macro copies data from the caller's "new" buffer into the kernel buffer. The `SYSCTL_OUT()` macro copies data from the kernel buffer into the caller's "old" buffer. These macros return zero if the copy is successful and an error number if it fails. In particular, if the caller buffer is too small, the macros will fail and return `ENOMEM`. These macros can be invoked multiple times. Each invocation advances an internal offset into the caller's buffer. Multiple invocations of `SYSCTL_OUT()` append the kernel buffers passed to the macro to the caller's "old" buffer, and multiple invocations of `SYSCTL_IN()` will read sequential blocks of data from the caller's "new" buffer.

One of the values returned to userland after a sysctl(3) invocation is the amount of data returned in the "old" buffer. The count is advanced by the full length passed to `SYSCTL_OUT()` even if the copy fails with an error. This can be used to allow userland to query the necessary length of an "old" buffer

Example 4

```
/*
 * 'arg1' points to the variable being exported, and 'arg2' specifies a
 * maximum value.  This assumes that negative values are not permitted.
 */
static int
sysctl_handle_int_range(SYSCTL_HANDLER_ARGS)
{
        int error, value;

        value = *(int *)arg1;
        error = sysctl_handle_int(oidp, &value, 0, req);
        if (error != 0 || req->newptr == NULL)
                return (error);
        if (value < 0 || value >= arg2)
                return (EINVAL);
        *(int *)arg1 = value;
        return (0);
}

static int foo;
SYSCTL_PROC(_debug, OID_AUTO, foo, CTLFLAG_RW | CTLTYPE_INT, &foo, 100,
    sysctl_handle_int_range, "I", "Integer between 0 and 99");

static int bar;
SYSCTL_PROC(_debug, OID_AUTO, bar, CTLFLAG_RW | CTLTYPE_INT, &bar, 0x100,
    sysctl_handle_int_range, "I", "Integer between 0 and 255");
```

for a node that returns a variable-sized buffer. If it is expensive to generate the data copied to the "out" buffer and a handler is able to estimate the amount of space needed, then the handler can treat this case specially. A caller queries length by using a `NULL` pointer for the "old" buffer. The handler can detect this case by comparing `req->oldptr` against `NULL`. The handler can then make a single call to `SYSCTL_OUT()` passing `NULL` as the kernel buffer and the total estimated length as the length. If the size of the data changes frequently, then the handler should overestimate the size of the buffer so that the caller is less likely to get an `ENOMEM` error on the subsequent call to query the node's state.

The `SYSCTL_OUT()` and `SYSCTL_IN()` macros can access memory in a user process. These accesses can trigger page faults if a user page is not currently mapped. For this reason, non-sleepable locks such as mutexes and reader/writer locks cannot be held when invoking these macros. Some control nodes return an array of state objects that correspond to a list of objects inside the kernel where the list is protected by a non-sleepable lock. One option such handlers can use is to allocate a temporary buffer in the kernel that is large enough to hold all of the output. The handler can populate the kernel buffer while it walks the list under the lock and then pass the populated buffer to `SYSCTL_OUT()` at the end after releasing the lock. Another option is to drop the lock around each invocation of `SYSCTL_OUT()` while walking the list. Some handlers may not want to allocate a temporary kernel buffer because it would be too large, and they may wish to avoid dropping the lock because the resulting races are too painful to handle. The system provides a third option for these handlers: the "old" buffer of a request can be wired by calling `sysctl_wire_old_buffer()`. Wiring the buffer guarantees that no accesses to the buffer will fault allowing `SYSCTL_OUT()` to be used while holding a non-sleepable lock. Note that this option is only available for the "old" buffer. There is no corresponding function for the "new" buffer. The `sysctl_wire_old_buffer()` function returns zero if it succeeds and an error number if it fails.

If a sysctl node wishes to work properly in a 64-bit kernel when it is acccessed by a 32-bit process, it can detect this case by checking for the `SCTL_MASK32` flag in `req->flags`. For example, a node that returns a long value should return a 32-bit integer in this case. A node that returns an array of structures corresponding to an internal list of objects may need to return an array of structures with an alternate 32-bit layout.

If a node allows the caller to alter its state via a "new" value, the handler should compare `req->newptr` against `NULL` to determine if a "new" value is supplied. A handler should only invoke `SYSCTL_IN()` and attempt to set a new value if `req->newptr` is non-`NULL`.

An example of a custom node handler that uses many of these features is the implementation of the "kern.proc.proc" node. The in-kernel implementation is more complex, but a simplified version is provided in Example 5.

Complex Branch Nodes

A branch node declared via `SYSCTL_NODE()` can specify a custom handler. If a handler is specified, then it is always invoked when any node whose address begins with the address of the branch node is accessed. The handler functions similarly to the custom handlers described above. Unlike `SYSCTL_PROC()`, the "arg1" and "arg2" parameters are not configurable. Instead, "arg1" points to an integer array containing the address of the node being accessed, and "arg2" contains the length of the address. Note that the address specified by "arg1" and "arg2" is relative to the branch node whose handler is being invoked. For example, if a branch node has the address 1.2 and node 1.2.3.4 is accessed, the handler for the branch node will be invoked with "arg1" pointing to an array containing "3, 4" and "arg2" set to 2. A simplified version of the "kern.proc.pid" handler is given below as Example 6. Recall that this is the node invoked by Example 1.

Dynamic Control Nodes

The control nodes described previously are static control nodes. Static nodes are defined in a source file with a fixed name and are created either when the kernel initializes the system control node subsystem or when a kernel module is loaded. Static nodes in a kernel module are removed when the kernel module is unloaded. The arguments passed to handlers for static nodes are also resolved at link time. This means that static nodes generally operate

Example 5

```
static int
sysctl_kern_proc_proc(SYSCTL_HANDLER_ARGS)
{
#ifdef COMPAT_FREEBSD32
        struct kinfo_proc32 kp32;
#endif
        struct kinfo_proc kp;
        struct proc *p;
        int error;

        if (req->oldptr == NULL) {
#ifdef COMPAT_FREEBSD32
                if (req->flags & SCTL_MASK32)
                        return (SYSCTL_OUT(req, NULL, (nprocs + 5) *
                            sizeof(struct kinfo_proc32)));
#endif
                return (SYSCTL_OUT(req, NULL, (nprocs + 5) *
                    sizeof(struct kinfo_proc)));
        }

        error = sysctl_wire_old_buffer(req, 0);
        if (error != 0)
                return (error);
        sx_slock(&allproc_lock);
        LIST_FOREACH(p, &allproc, p_list) {
                PROC_LOCK(p);
                fill_kinfo_proc(p, &kp);
#ifdef COMPAT_FREEBSD32
                if (req->flags & SCTL_MASK32) {
                        freebsd32_kinfo_proc_out(&kp, &kp32);
                        error = SYSCTL_OUT(req, &kp32, sizeof(kp32));
                } else
#endif
                        error = SYSCTL_OUT(req, &kp, sizeof(kp));
                PROC_UNLOCK(p);
                if (error != 0)
                        break;
        }
        sx_sunlock(&allproc_lock);
        return (error);
}
SYSCTL_PROC(_kern_proc, KERN_PROC_PROC, proc, CTLFLAG_RD |
    CTLFLAG_MPSAFE | CTLTYPE_STRUCT, NULL, 0, sysctl_kern_proc_proc,
    "S,kinfo_proc", "Process table");
```

on global variables.

The kernel also provides support for dynamic control nodes. Unlike static nodes, dynamic nodes can be created or destroyed at any time. They can also use dynamically generated names and reference dynamically allocated variables. Dynamic nodes can be created as new children of both static and dynamic nodes.

`sysctl` Contexts

To safely remove dynamic control nodes, each node must be explicitly tracked and removed in a safe order (leaves before branches). Doing this by hand is tedious and error prone, so the kernel provides a sysctl context abstraction. A sysctl context is a container that tracks zero or more dynamic control nodes. It allows all of the control nodes it contains to be safely removed in an atomic transaction.

The typical practice is to create one context for each group of related nodes via a call to `sysctl_ctx_init()`. All of the nodes are added to the context during initialization (such

as when a driver attaches to a device). Only a reference to the context has to be maintained. A single call to `sysctl_ctx_free()` during teardown (such as when a driver detaches from a device) is sufficient to remove the entire group of control nodes.

Adding Dynamic Control Nodes

Dynamic control nodes are created by using one of the `SYSCTL_ADD_*()` macros from `<sys/sysctl.h>`. Each of these macros corresponds to a macro used to create static node with the following differences:

- The dynamic macros are invoked within a code block rather than at the top level. The dynamic macros return a pointer to the created node.
- The dynamic macros add an additional argument that is a pointer to the sysctl context the new node should be associated with. This is given as the first argument to the macro.
- The parent argument is specified as a pointer to the node list belonging to the parent node. Two helper macros are provided to locate these pointers. The `SYSCTL_STATIC_CHILDREN()` macro should be used when the parent node is a static control node. It takes the parent node's name as the sole argument. The name is formatted in the same manner that a parent is specified when declaring a static node. For parents that are dynamic nodes, the `SYSCTL_CHILDREN()` macro should be used instead. It accepts a pointer to the parent node as returned by a previous invocation of `SYSCTL_ADD_NODE()` as its sole argument.
- The name argument is specified as a pointer to a C string rather than an unquoted identifier. The kernel will create a duplicate of this string to use as the name of the node. This allows the name to be constructed in a temporary buffer if needed.
- The kernel will also create a duplicate of the description argument so that it can be constructed in a temporary buffer if needed.

Example 7 defines two functions to manage a dynamic sysctl node. The first function initializes a sysctl context and creates the new node. The second function destroys the node and destructs the context.

Tunables

Another kernel API that is often used in conjunction with control nodes is the tunable API. Tunables are values stored in the kernel's environment. This environment is populated by the boot loader and can also be modified at run-time by the kenv(1) command. The kernel environment consists of named variables with string values similar to the environment of user processes. The API provides two sets of macros in `<sys/kernel.h>`.

```
static int
sysctl_kern_proc_pid(SYSCTL_HANDLER_ARGS)
{
        struct kinfo_proc kp;
        struct proc *p;
        int *mib;

        if (arg2 == 0)
                return (EISDIR);
        if (arg2 != 1)
                return (ENOENT);
        mib = (int *)arg1;
        p = pfind(mib[0]);
        if (p == NULL)
                return (ESRCH);
        fill_kinfo_proc(p, &kp);
        PROC_UNLOCK(p);
        return (SYSCTL_OUT(req, &kp,
            sizeof(kp)));
}
static SYSCTL_NODE(_kern_proc, KERN_PROC_PID,
    pid, CTLFLAG_RD | CTLFLAG_MPSAFE,
    sysctl_kern_proc_pid,
    "Process information");
```

Example 6

The first set (`TUNABLE_*()`) are declared at the top-level similar to static control nodes and fetch the value of a tunable at either boot time or when a module is loaded. The second set of macros (`TUNABLE_*_FETCH()`) can be used in a code block to fetch a tunable at runtime. Each macro accepts a C string name as the first argument that specifies the name of the tunable to read. When using tunables in conjunction with control nodes, the convention is to use the name of the control node as the tunable's name.

The tunable API supports several integer types. Each macro accepts a pointer to an integer variable of the corresponding type as the second argument. Each macro invocation searches the kernel's environment for the requested tunable. If the tunable is found and the entire string value is parsed successfully, the integer variable is changed to

SUFFIX	VALUE
k	2^10
m	2^20
g	2^30
t	2^40

Table 4

Example 7

```
static struct sysctl_ctx_list ctx;

static int
load(void)
{
        static int value;
        int error;

        error = sysctl_ctx_init(&ctx);
        if (error)
                return (error);
        if (SYSCTL_ADD_INT(&ctx, SYSCTL_STATIC_CHILDREN(_debug), OID_AUTO,
            "dynamic", CTLFLAG_RW, &value, 0, "An integer") == NULL)
                return (ENXIO);
        return (0);
}

static int
unload(void)
{

        return (sysctl_ctx_free(&ctx));
}
```

the parsed value. Note that overflows are silently ignored. If the tunable is not found or contains invalid characters, the integer variable is left unchanged. The macros provided for integers are: `TUNABLE_INT()` for signed integers, `TUNABLE_LONG()` for signed long integers, `TUNABLE_ULONG()` for unsigned long integers, and `TUNABLE_QUAD()` for signed 64-bit integers.

The string value of an integer tunable is parsed in the same manner as strtol(3) with a base of zero. Specifically, a string that begins with "0x" is interpreted as a hexadecimal value, a string that begins with "0" is interpreted as an octal value, and all other strings are interpreted as a decimal value. In addition, the string may contain an optional single character suffix that specifies a unit. The value is scaled by the size of the unit. The unit is case-insensitive. Supported units are described in Table 4.

String tunables are also supported by the `TUNABLE_STR()` macro. This macro accepts three arguments: the name of the tunable, a pointer to a character buffer, and the length of the character buffer. If the tunable does not exist in the kernel environment, the character buffer is left unchanged. If the tunable does exist, its value is copied into the buffer. The string in the buffer is always terminated with a null character. The value will be truncated if it is too long to fit into the buffer.

The `TUNABLE_*_FETCH()` macros accept the same arguments as the corresponding `TUNABLE_*()` macro. They also have the same semantics with one additional behavior. These macros return an integer value of zero if the tunable is found and successfully parsed, and non-zero otherwise.

System control nodes that have a corresponding tunable should use either the `CTLFLAG_RDTUN` or `CTLFLAG_RWTUN` flag to specify the allowed access to the node. Note that this does not cause the system to implicitly fetch a tunable based on the node's name. The tunable must be fetched explicitly. However, it does provide a hint to the sysctl(8) utility that is used in diagnostic messages.

Example 8 demonstrates the use of a tunable in a device driver to fetch a default parameter. The parameter is available as a read-only control node that can be queried by the user (this is helpful for the user when determining the default value). It also includes a portion of the attach routine where the global tunable is used to set the initial value of a per-device control variable. A dynamic sysctl is created for each device to allow the variable to be changed for each device independently. The sysctl is stored in the per-device sysctl tree

Example 8

```
static SYSCTL_NODE(_hw, OID_AUTO, foo, CTLFLAG_RD, NULL,
    "foo(4) parameters");

static int foo_widgets = 5;
TUNABLE_INT("hw.foo.widgets", &foo_widgets);
SYSCTL_INT(_hw_foo, OID_AUTO, widgets, CTLFLAG_RDTUN, &foo_widgets, 0,
   "Initial number of widgets for each foo(4) device");

static int
foo_attach(device_t dev)
{
        struct foo_softc *sc;
        char descr[64];

        sc = device_get_softc(dev);
        sc->widgets = foo_widgets;
        snprintf(descr, sizeof(descr), "Number of widgets for %s",
            device_get_nameunit(dev));
        SYSCTL_ADD_INT(device_get_sysctl_ctx(dev),
            SYSCTL_CHILDREN(device_get_sysctl_tree(dev)), OID_AUTO,
            "widgets", CTLFLAG_RW, &sc->widgets, 0, descr);
        ...
}
```

created by the new-bus subsystem. It also uses the per-device sysctl context so that the sysctl is automatically destroyed when the device is detached.

The interface for system control nodes is defined in `<sys/sysctl.h>` and the implementation can be found in `sys/kern/kern_sysctl.c`. It may be particularly useful to examine the implementation of the predefined handlers. First, they demonstrate typical uses of `SYSCTL_IN()` and `SYSCTL_OUT()`. Second, they can be used to marshal data in custom handlers.

The interface for tunables is defined in `<sys/kernel.h>` and the implementation can be found in `sys/kern/kern_environment.c`.

John Baldwin joined the FreeBSD Project as a committer in 1999. He has worked in several areas of the system, including SMP infrastructure, the network stack, virtual memory, and device driver support. John has served on the Core and Release Engineering teams and organizes an annual FreeBSD developer summit each spring.

FreeBSD and Commercial Workloads:

Managed Services at NYI

BY JOSEPH KONG

This white paper examines FreeBSD's role in commercial workloads. We discuss the challenges associated with running an ISP, specifically managed services, and we present some of the unique solutions that FreeBSD provides. To help present this material we profile NYI, an ISP headquartered in New York. NYI provides colocation, dedicated servers, web and email hosting, managed services, turnkey disaster recovery, and business continuity solutions. It specializes in mission-critical data services for the financial, architectural, fashion, law, life sciences, media, and real estate industries.

Managed services is the practice of outsourcing day-to-day management responsibilities as a method for improving operations. Using managed services in the information technology (IT) sector, organizations can avoid the burden of maintaining equipment and infrastructure, thereby allowing their IT staff to focus on core business tasks instead. Managed services providers essentially offer the following benefits:

- **Ensure reliability**—by keeping networks up and running and minimizing downtime. This includes defending against malware.
- **Stay up-to-date**—by keeping hardware and software updated, as well as keeping pace with bandwidth demands.

Challenges and Solutions

There are a number of challenges associated with providing managed services at NYI, including interacting with the wider Internet, ensuring high availability, recovering from data loss, and compartmentalizing systems and data. Each challenge is surmounted with a FreeBSD-based solution.

Outsider Threats (or the Internet at Large)

IT security and malware threats have been steadily increasing. According to Symantec, "the release rate of malicious code and other unwanted programs may be exceeding that of legitimate software applications."[2] A firewall is now the de facto standard for protecting against threats from the public Internet. However, on most networks the firewall is a single point of failure. When the firewall goes down, access to and from the internal network comes to a halt, effectively creating downtime for your customers. FreeBSD provides three components—PF, CARP, and pfsync—which NYI uses in tandem to keep their systems well protected with zero downtime.

The Packet Filter system (PF) is the firewall. NYI employs at least two firewalls in parallel, where one of the firewalls is the primary and the rest are the backups. All traffic passes through the primary, and if the primary ever fails, a backup will assume the identity of the primary and continue where it left off. Existing connections are preserved and traffic continues as if nothing happened. An addi-

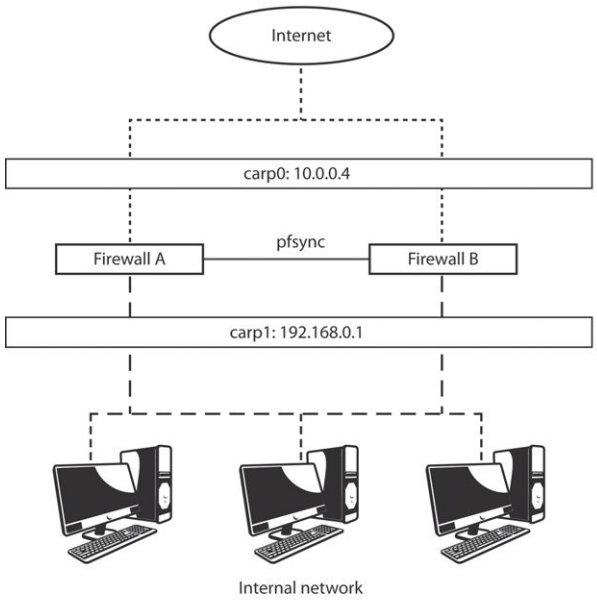


Figure 1: A basic PF, CARP, and pfsync setup.[4]

tional benefit of this configuration is that it is trivial to do maintenance and upgrades on the firewalls without impacting the network. Simply take the firewalls offline one at a time.

The Common Address Redundancy Protocol (CARP) is what allows a backup firewall to assume the identity of the primary. The primary purpose of CARP is to allow multiple hosts on the same network segment to share an IP address.[3] Within each CARP group, the primary firewall, known as the master, holds the shared IP address. It responds to any traffic or Address Resolution Protocol (ARP) requests directed toward it. The primary firewall regularly sends out CARP advertisements, and the backups listen for this advertisement. If the backups don't hear an advertisement from the primary for a set period of time, they will begin sending their own advertisements. The backup that advertises most frequently will become the new primary.

pfsync is the system that synchronizes the firewalls' state tables. This system is how a backup firewall can preserve the connections of the primary, when the primary fails. The primary firewall sends out pfsync messages containing its state information. To ensure that every backup firewall is synchronized, the backup firewalls replicate these messages and send them out too.

In Figure 1, the two firewalls (A and B) each have three network interfaces. The interface on the 10.0.0.0/24 subnet connects to the external network (that is, the Internet). The interface on the 192.168.0.0/24 subnet connects to the

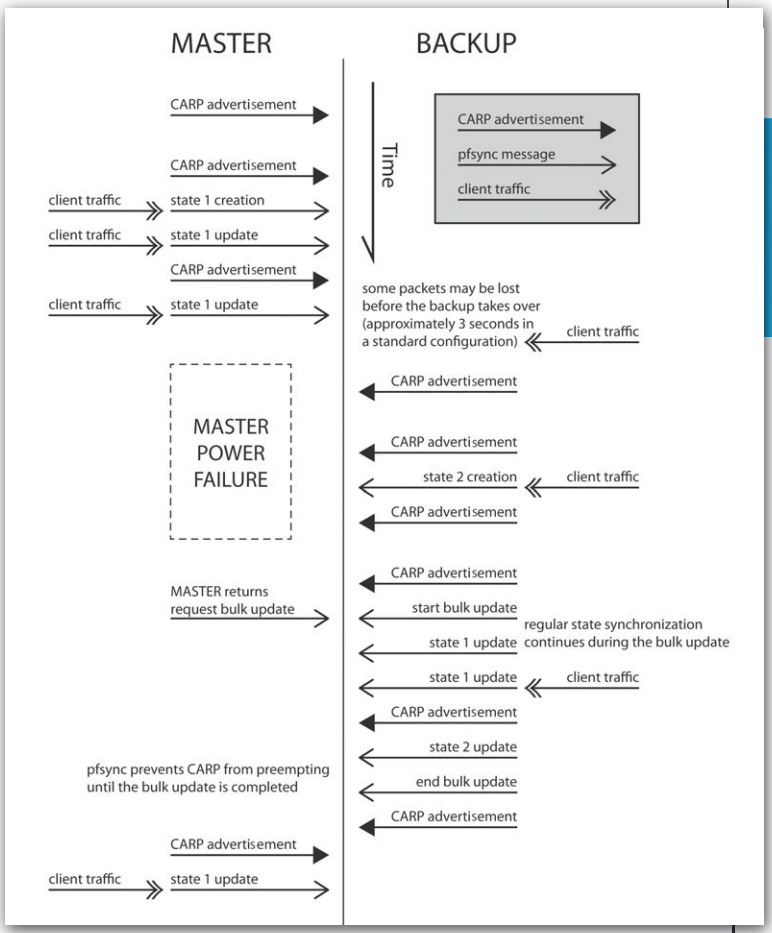


Figure 2: Timeline of events in a typical failover.[5]

internal network. Finally, the third interface connects the two firewalls to each other via a crossover cable, which forms a dedicated link for the pfsync messages. Figure 2 provides the timeline of events in a typical failover scenario.

High Availability

Round-the-clock operations have long been a requirement in the ISP industry. With customers spread globally across numerous time zones, any interruption of service, at any time, will affect customers, and a customer who cannot access an online system will inevitably be dissatisfied. To ensure high availability of services, NYI makes use of HAProxy, which is available in the FreeBSD ports tree, with CARP (which was discussed in the previous section).

HAProxy is a TCP/HTTP load balancer, which is used to improve the performance of web sites and web services by spreading the incoming requests across multiple web servers, thereby ensuring that no individual server is overburdened.

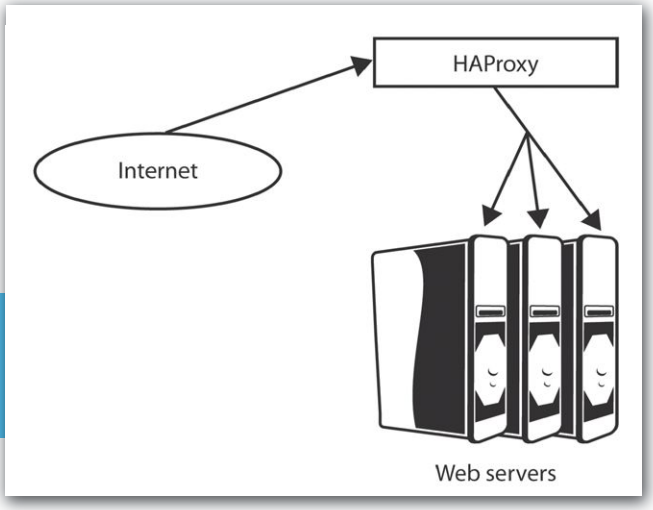


Figure 3: Demonstrating how HAProxy works.[6]

In Figure 3, HAProxy accepts a request from the external network (that is, the Internet) and forwards it to the least-used web server within the internal network.

CARP, as mentioned previously, is what allows a backup system to assume the identity of a primary. To ensure high availability for its three largest managed setups (*Men's Journal, Rolling Stone*, and *Us Magazine*), NYI employs a pair of machines running HAProxy with CARP. If the primary load-balancing machine fails, the backup will assume the identity of the primary and take over.

From this example, we can see that CARP can provide failover redundancy for systems beyond just firewalls. As another example, NYI employs CARP with some of its managed Internet Protocol Security (IPsec) virtual private networks (VPNs).

Disaster Recovery

Backup and data recovery have long been standard data center disciplines and are equally important for providers of managed services. Any data loss has the potential to significantly impact the profitability of a company. NYI takes advantage of FreeBSD's GEOM mirroring to mitigate this risk.

GEOM mirroring is FreeBSD's way of implementing RAID 1, which creates reliable data storage by generating an exact copy (or mirror) of a data set on two or more disk drives. When a drive fails, the data remains available because it can be provided by the other functioning drives, allowing administrators to replace the failed drive without interrupting their users.

One interesting feature of GEOM mirroring is that it can also be used to quickly clone servers. At NYI the process is as follows:

1. Remove a drive from the mirror.
2. Execute fsck(8) on the drive; this checks the consistency and repairs any damaged file systems on the drive.
3. Mount the drive in order to adjust any settings; mounting a drive makes it accessible through the operating system's file system.
4. Adjust settings as needed.
5. Unmount the drive.
6. Put the drive into a new server.

Steps three through five can be omitted if no settings need to be adjusted. In addition to GEOM mirroring, NYI uses FreeNAS, ZFS, and rsync to perform offsite backups in order to mitigate the risk of data loss.

FreeNAS is based on an embedded version of FreeBSD and provides an open source network-attached storage (NAS) solution. NAS systems provide data storage to other devices on a network and communicate in terms of files, rather than in disk blocks.

In Figure 4, the end users read and write files to the NAS system over an Ethernet network. NYI employs multiple FreeNAS machines with over 20 TB (terabytes) of storage to house offsite backups.

ZFS is the file system used by FreeNAS (and optionally by FreeBSD). Its features include support for high storage capacities, protection against data corruption, continuous integrity checking, automatic repair of data, software raid

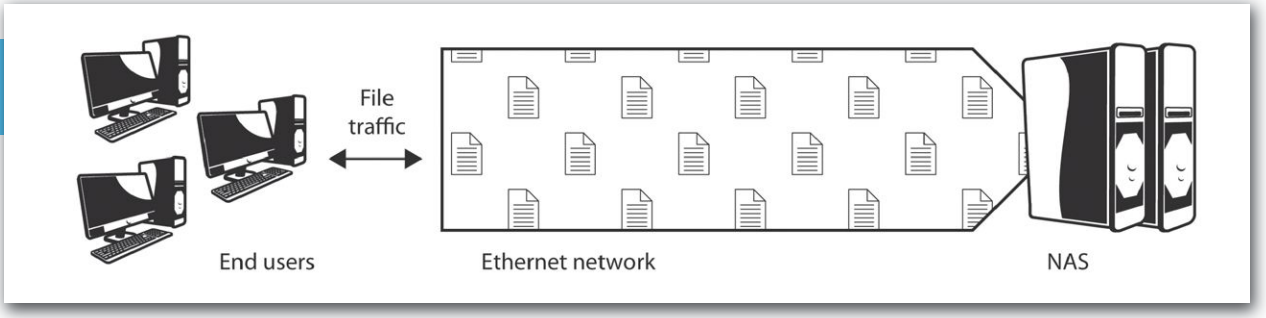


Figure 4: A basic NAS setup.[7]

(RAID-Z), instantaneous file system snapshots, and more. In short, ZFS is designed for data integrity from top to bottom, which is desirable when managing backups.

rsync is the network protocol that NYI uses to back up a machine's entire file system offsite to a FreeNAS machine. rsync minimizes data transfer by using delta encoding, which transmits data in the form of differences rather than complete files. After the first full backup, rsync will only transfer the differences between the local and backed-up copy.

Compartmentalization

The desire to establish a clean and clear-cut separation between services, for security purposes, has always been a challenge for system administrators. Traditional Unix systems provide chroot(2); however, chroot(2) has a number of limitations (for example, it does not defend against intentional tampering by the root user). FreeBSD has modified and improved on the traditional chroot(2) concept with jails.

FreeBSD jails compartmentalize the system. Each jail is a virtual environment running on the host with its own set of files, processes, users, and root user. Unlike a chroot(2) environment, which restricts processes to a particular view of the file system, jails restrict what a process can do in relation to the rest of the system. Jailed processes are sandboxed.[8]

Here is an example usage for jails: A small-scale managed customer of NYI, Expand the Room, requested a staging and production environment for a website they were developing. The solution was a FreeBSD machine with two jails that were identical in every way except for IP address and hostname.

NYI also uses jails internally for hardware-efficient Domain Name System (DNS) servers. For example, a FreeBSD machine may contain a jail for recursive DNS, a second jail for authoritative customer DNS, and a third jail for authoritative NYI DNS. Each of these jailed DNS servers then uses CARP within a cluster of machines to ensure failover redundancy. Figure 5 demonstrates this.

In Figure 5, two servers (A and B) are used to provide three distinct services. Each service is contained within its own jail (jail0, jail1, or jail2) and uses CARP to ensure high availability.

Managed Services at NYI

Customers

Managing customer expectations is always a challenge. Customers expect and demand things to just work with (near) 100% uptime. This is exacerbated by the fact that managed services providers cannot control the client software that their customers employ, which makes compatibility an issue. FreeBSD's open source nature helps address this challenge.

For example, one of NYI's largest managed customers used a particularly buggy SSH client, which expected the challenge-response password prompt to be "Password: " (note the space). However, the prompt in FreeBSD is "Password:" (without a space after the colon) and this caused the SSH client to fail at authenticating. Since FreeBSD is open source, NYI could easily patch FreeBSD's SSH server to include a space in the password prompt, allowing their customer to continue using the client of their choice.

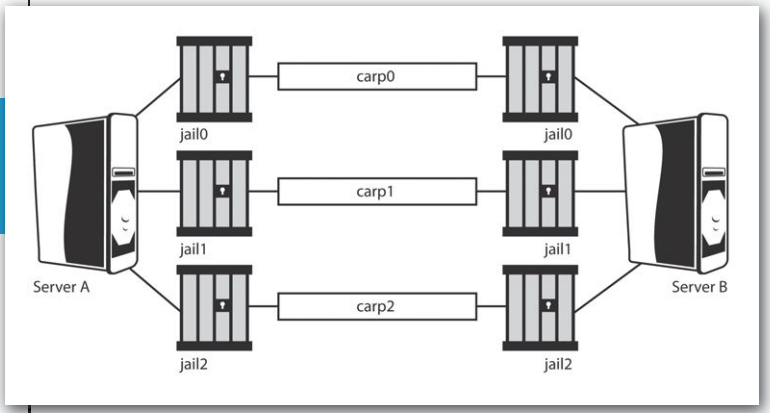


Figure 5: Jailed services with CARP.

About NYI

Established in 1996, NYI is headquartered in the heart of Wall Street. Its core services include colocation, dedicated servers, web and email hosting, managed services, turnkey disaster recovery, and business continuity solutions. NYI owns and maintains its own data centers and with its high-bandwidth connectivity partners (Zayo, Verizon Business, Optimum Lightpath, AT&T, Level 3, and GTT), NYI specializes in mission-critical data services for the financial, architectural, fashion, law, life sciences, media, and real estate industries. NYI is SSAE 16 Type II-compliant as well as being PCI and HIPAA compliant. For more information about NYI, visit www.nyi.net.

The FreeBSD Foundation is a 501(c)(3) nonprofit organization dedicated to supporting the FreeBSD Project. The Foundation gratefully accepts donations from individuals and businesses, using them to fund projects that further the development of the FreeBSD operating system. In addition, the Foundation can represent the FreeBSD Project in executing contracts, license agreements, and other legal arrangements that require a recognized legal entity. The FreeBSD Foundation is entirely supported by donations. For more information about the Foundation visit www.freebsdfoundation.org.

Conclusion

In 1996, when NYI was founded, FreeBSD was the only viable open source Unix. Today, FreeBSD continues to drive NYI for the reasons outlined in this article and more. Boris Kochergin, NYI's Chief Rigor Officer, had these additional things to say about why NYI uses FreeBSD:

"FreeBSD has excellent documentation. The FreeBSD Handbook, which covers the day-to-day use of FreeBSD, is clear, concise, and provides an easy means for administrators to learn the system. FreeBSD is open source and the code is well organized, so it's easy and possible to fully understand it. Finally, FreeBSD continues the BSD legacy of empowering the Internet!"[9] ●

Joseph Kong is a self-taught computer enthusiast who dabbles in the fields of exploit development, reverse code engineering, rootkit development, and systems programming (FreeBSD, Linux, and Windows). He is the author of the critically acclaimed Designing BSD Rootkits and FreeBSD Device Drivers. For more information about Joseph Kong visit www.thestackframe.org or follow him on Twitter @JosephJKong.

FOOTNOTES

1. "Managed services," last modified August 01, 2013, http://en.wikipedia.org/wiki/Managed_services.

2. "Symantec Internet Security Threat Report: Trends for July–December '07," April, 2008, p. 29.

3. "PF: Firewall Redundancy with CARP and pfsync," last modified May 01, 2013, http://www.openbsd.org/faq/pf/carp.html.

4. Figure 1 is adapted from "Firewall Failover with pfsync and CARP," accessed August 14, 2013, http://www.countersiege.com/doc/pfsync-carp/.

5. Figure 2 is adapted from "Firewall Failover with pfsync and CARP," accessed August 14, 2013, http://www.countersiege.com/doc/pfsync-carp/.

6. Figure 3 is adapted from "HAProxy," accessed August 14, 2013, http://haproxy.1wt.eu/.

7. Figure 4 is adapted from "Big data meets big storage," accessed August 14, 2013, http://arstechnica.com/business/2011/05/isilon-overview/2/.

8. "FreeBSD jail," last modified June 08, 2013, http://en.wikipedia.org/wiki/FreeBSD_jail.

9. The first widely-used TCP/IP implementation was from BSD.

FreeBSD® JOURNAL

LEAK THIS!

FreeBSD® Journal is published by The FreeBSD Foundation

GETTING STARTED with FreeBSD on BeagleBone BLACK

The BeagleBone Black (BBB) is a $45 PC that fits in an Altoids tin. It is built on the same Texas Instruments "Sitara" chip as the earlier BeagleBone (which had a white circuit board), but is now much cheaper and significantly more capable.

BY TIM KIENTZLE

As you can see from the BeagleBoard.org website, the potential of the BeagleBone Black (BBB) is immense. The sidebar below shows a few examples of possible applications: Best of all, the BBB can run a standard FreeBSD system with all the tools and support you've come to expect.

FreeBSD on BBB

Popular new ARM systems such as the BeagleBone and Raspberry Pi have generated a lot of developer interest in FreeBSD/ARM. In the last year, most parts of FreeBSD—boot loaders, kernel, toolchain, drivers, userland, and ports—have seen significant improvement on ARM platforms.

Right now, the development version of FreeBSD supports the BBB reasonably well:

- Serial console (requires an adapter cable similar to Adafruit #954).
- Full FreeBSD boot loader, including boot-time module loading and Forth scripting.
- Device-tree based kernel.
- Micro-SD and eMMC support.
- USB host support.
- Experimental USB client support (the BBB can act like a USB device).
- 10/100 Ethernet.
- FreeBSD/ARM now uses clang as the default compiler.
- FreeBSD/ARM now uses the EABI calling convention, which offers slightly better performance and better compatibility with other compilers; if you have binaries or libraries that were compiled before this change, you will have to recompile them.
- FreeBSD can rebuild and upgrade natively on the BBB.
- A growing number of ports build and run on the BBB.

Author's Warning: I'm writing this in September 2013 based on the current status of the FreeBSD development branch. Much of the following will have changed by the time FreeBSD10 is finally released. Ask on the FreeBSD/ARM or FreeBSD current mailing lists for more up-to-date information.

Possible Applications

- **MICRO-SERVER:** The 1GHz ARM Cortex A8 processor, 512MB of RAM, 10/100 Ethernet, and 2GB of flash is plenty to run a personal or small-office web or mail server.
- **EMBEDDED:** Embedded: The BBB includes extensive GPIO and hardware expansion support and requires only 2.5 watts for basic operation. (USB requires additional power.)
- **EDUCATION:** The low price and expandability make BBB a good choice for learning about software and hardware development.

There are a few areas that still need improvement, however.

- The FreeBSD package team has plans for a public ARM package repository, but it is not yet available.
- Video and audio drivers have yet to be written.
- Expansion capes are not supported.

Your First FreeBSD Boot

To boot FreeBSD, you first build a micro-SD card with FreeBSD installed, and then boot the BBB from the micro-SD card.

What you'll need:

- BeagleBone Black.
- 5v power supply or Mini-USB cable.
- Micro-SDHC card 4GB or larger.
- Serial cable such as Adafruit #954 or FTDI TTL-232R-3V3 (optional but highly recommended).

1. Build or Download a FreeBSD Image

Below, I'll explain how you can build your own FreeBSD image. To get started, you can download an image from the FreeBSD.org website:

ftp://ftp.freebsd.org/pub/FreeBSD/snapshots/
http://ftp.freebsd.org/pub/FreeBSD/snapshots/

Caveat: "Snapshot" images are built from whatever source happened to be current in the FreeBSD development branch that day. Stable images will be released as soon as FreeBSD 10 is finalized, which is expected to occur before the end of 2013. Downloaded images are usually compresssed; you'll need to uncompress yours before you can copy it onto a micro-SD card.

2. Copy onto a Micro-SD Card

The 'dd' utility is used for raw data copies such as initializing a disk from a raw image. You first need to connect the SD card to your computer (likely using some sort of adapter) and identify the correct device name.

The easiest way to do this is to

```
$ ls -l /dev
```

before connecting the SD card.

Then do

```
$ ls -l /dev
```

again after you've connected it; the new entry should be obvious.

If the SD card is already formatted, you'll see several entries appear for each partition on the card. Since we want to overwrite the whole card, you need to identify the base device, which is generally a couple of letters followed by a single digit (e.g., "da7", "mmc4", or "sdhci0").

For this task, you want to provide three arguments to the 'dd' command: the input file (if), the output device (of), and the block size to use when copying (bs). You don't have to specify a block size, but the default setting results in very slow operation.

For example, if your micro-SD is connected as 'da7', then the full command will look like this:

```
$ dd if=FreeBSD-BeagleBone.img
of=/dev/da7 bs=8m
```

Depending on how your system security is set up, you will probably have to run this command as root using 'sudo' or similar.

Once the micro-SD card is imaged, you can insert it into the BBB.

3a. (Optional but Recomended) Attach Serial Cable

Once you have the SD card built, you're ready to hook up the BeagleBone Black and boot FreeBSD. Since FreeBSD doesn't yet support the HDMI output on the BBB, you should consider using a serial cable so you can see what's going on. Without a serial cable, you can wait until it boots and try to connect over SSH, but it's much harder to diagnose if anything goes wrong.

The BBB has a low-voltage serial interface that requires a special adapter cable. Make certain you are using a 3.3v adapter, since similar cables come in 5v and 1.8v versions that will not work with BBB (Figure 1).

3b. Open a Terminal Window

The serial adapter is powered by USB from the host system, so it starts working as soon as it is

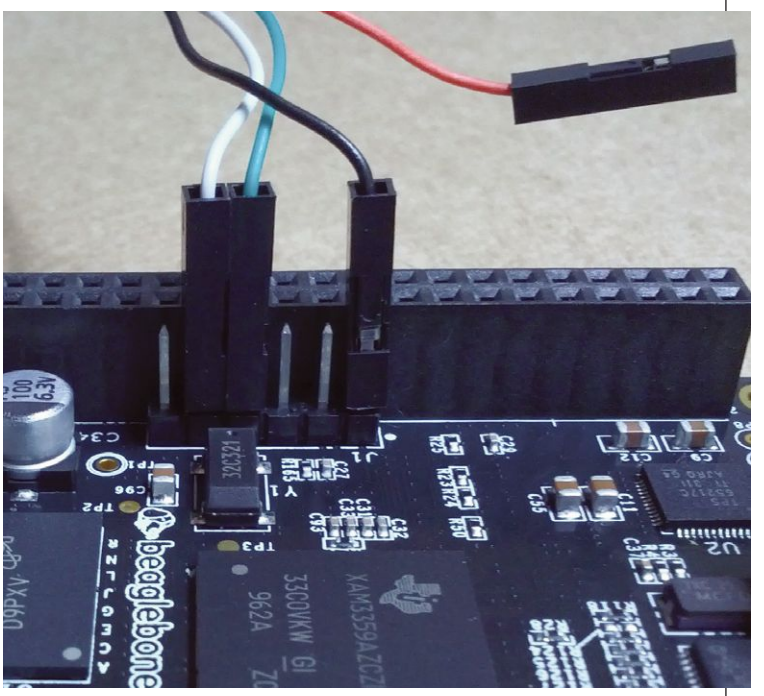


Figure 1. Adafruit #954 serial cable connected to BBB.

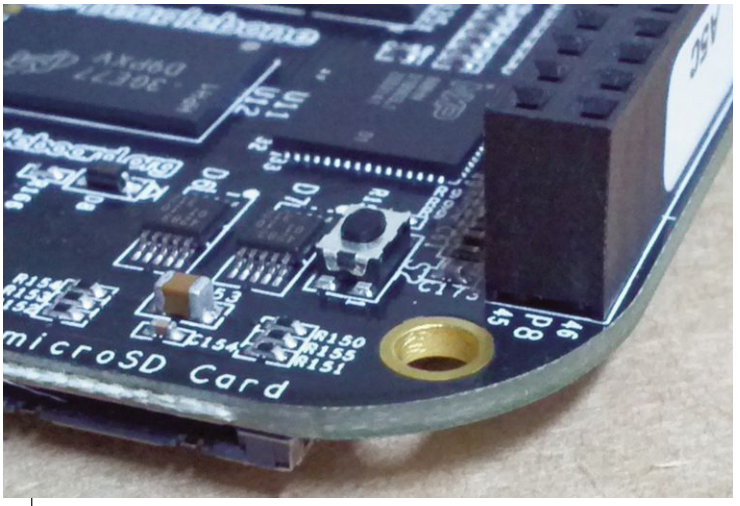


Figure 2. The boot switch is just above the micro-SD slot.

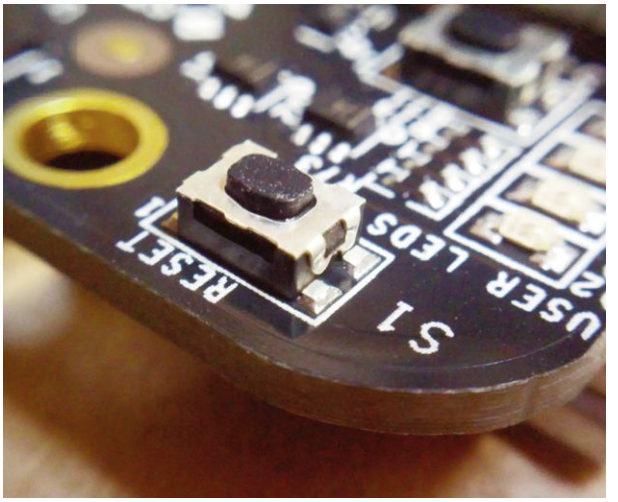


Figure 3. The reset switch is in the corner of the board at the Ethernet adapter end.

plugged into the USB, even before the BBB has power.

To use it from FreeBSD, use the 'cu' utility, specifying the line speed of 115200 baud and the appropriate "tty" device:

```
$ sudo cu -s 115200 -l /dev/ttyU0
```

Of course, you won't see anything until you actually apply power.

4. Hold the Boot Switch and Apply Power

At this point, you should NOT have any power connected to your BBB. If you've already connected a 5v power supply or a mini-USB cable, then unplug it and read the following carefully.

(The detailed logic for when the BBB boots from eMMC or micro-SD is a little complicated. I've been confused many times when the BBB booted from the wrong source.)

The "boot switch" determines whether the BBB boots from eMMC (the default) or from micro-SD (Figure 2).

To boot from micro-SD reliably, you must:

* Hold down the boot switch
* Apply power
* Count to 3
* Release the boot switch

The BBB power chip remembers the boot switch status, so it will continue to boot and reboot from micro-SD until you disconnect the power supply entirely.

Hint: If you need to reboot, leave the power connected and tap the reset switch (Figure 3), which will reboot from the same source.

Hint: If you get random shutdowns and are powering with a mini-USB cable, try getting a separate 5v power supply. The BBB power requirements are just at the edge of what standard USB ports will provide.

Hint: If you see the four LEDs start flashing rapidly, you've booted the Linux image from eMMC. Remove power, hold the boot switch, and try again.

What You Should See When You Boot

If you're familiar with how FreeBSD boots on i386 or amd64 PCs, then the BBB boot process will look very familiar, although there are a couple of differences. Most obviously, the initial boot stages are handled by "U-Boot", a GPL boot loader project that supports a wide variety of hardware.

1. MLO/SPL: U-Boot First Stage

When the TI Sitara chip first initializes, it does not have access to the main RAM. As a result, the very first boot stage must fit into 128k of on-chip memory.

```
U-Boot SPL 2013.04
(Aug 03 2013 - 21:27:30)
OMAP SD/MMC: 0
reading bb-uboot.img
reading bb-uboot.img
```

U-Boot provides a small program called SPL which the TI Sitara chip loads from a file called "MLO". This program is just enough to initialize the DRAM chip and load the main U-Boot program from the micro-SD card.

2. U-Boot Main Loader

U-Boot is a GPL-licensed boot loader that supports a wide variety of hardware. Although originally developed for Linux, U-Boot's robust hardware support, scriptability, and active community make it a good choice for booting FreeBSD as well.

U-Boot starts by initializing the USB, network, and MMC/SD interfaces.

```
U-Boot 2013.04 (Aug 03 2013 -
21:27:30)
  ... other messages ...
reading bb-uEnv.txt
reading bbubldr
240468 bytes read in 33 ms (6.9 MiB/s)
reading bboneblk.dtb
14210 bytes read in 7 ms (1.9 MiB/s)
Booting from mmc ...
## Starting application at 0x88000054
...
```

Once it has the MMC/SD initialized, it reads three files into memory.

* bb-uEnv.txt is empty by default, but you can edit this to redefine the U-Boot startup functions.

* bbubldr is the FreeBSD boot loader that will be run next.

* bboneblk.dtb is the DTB file described below.

3. About the DTB File

Operating systems for newer embedded processors are increasingly using a "device tree" file—sometimes called a "flattened device tree" (fdt)—to initialize the kernel. This file lists all the peripherals and helps the kernel decide which drivers to enable. Device trees are compiled: The source version is called DTS and the binary compiled version is called a DTB file.

The U-Boot initialization checks which hardware you are currently running and then loads the appropriate DTB file into memory. This data is not directly used by U-Boot or by ubldr, but is eventually passed to the FreeBSD kernel. The key advantage of this arrangement: The exact same kernel can run on both BeagleBone and BeagleBone Black since key configuration such as the amount of RAM and number of drives is provided by the DTB.

Eventually, the FreeBSD/ARM developers hope to have a single GENERIC kernel that boots on a number of boards. This requires more work on the kernel to ensure that the various board support routines can coexist. It also requires more work on the boot loader side to ensure that all of the various loaders correctly provide a DTB file to the kernel.

4. FreeBSD Ubldr

U-Boot knows a lot about the BBB hardware and how to initialize it, but does not know anything about the FreeBSD kernel and modules.

So the BBB uses U-Boot to load "ubldr". This is essentially the same as "BTX loader" used to boot FreeBSD on i386/amd64, but with a few changes so that it works with U-Boot instead of the PC BIOS (hence the name "ubldr" for "U-Boot compatible LoaDeR").

```
Consoles: U-Boot console
Compatible API signature found
@8f246240
Card did not respond to voltage
select!
Number of U-Boot devices: 2
FreeBSD/armv6 U-Boot loader, Revision
1.2
(root@fci386.localdomain, Fri Aug 16
12:59:51 PDT 2013)
DRAM:  256MB
Device: disk
Loading /boot/defaults/loader.conf
/boot/kernel/kernel
data=0x449864+0x17d3c8
syms=[0x4+0x82890+0x4+0x4ec85]
Hit [Enter] to boot immediately, or
any other key for command prompt.
Booting [/boot/kernel/kernel]...
Using DTB provided by U-Boot.
Kernel entry at 0x80200100...
Kernel args: (null)
```

5. Load loader.rc, loader.conf

Ubldr pulls in a lot of standard FreeBSD configuration. In particular, it reads loader.conf and possibly loader.rc. These can be used to load kernel modules into memory so they are available when the kernel first boots.

6. Load FreeBSD Kernel

Ubldr can now load the FreeBSD kernel proper.

7. Start FreeBSD Kernel

Once everything is ready, ubldr actually starts the FreeBSD kernel. The last lines printed by ubldr indicate how it is going to launch the kernel:

```
Booting [/boot/kernel/kernel]...
Using DTB provided by U-Boot.
Kernel entry at 0x80200100...
Kernel args: (null)
```

8. Initialize FreeBSD Kernel

Unlike ubldr, which relies heavily on U-Boot, the FreeBSD kernel runs completely on its own.

So it must first set up its own memory management and console handling. Once that is done, the kernel can show its first message:

```
KDB: debugger backends: ddb
KDB: current backend: ddb
Copyright (c) 1992-2013 The FreeBSD
Project.
Copyright (c) 1979, 1980, 1983, 1986,
```

```
1988, 1989, 1991, 1992, 1993, 1994
      The Regents of the University
of California. All rights reserved.
FreeBSD is a registered trademark of
The FreeBSD Foundation.
FreeBSD 10.0-CURRENT #0 r254265: Fri
Aug 16 12:58:43 PDT 2013
root@fci386.localdomain:/usr/..../src/
sys/BEAGLEBONE arm
```

The kernel then proceeds to use the device tree data to identify each system that needs to be initialized.

9. Start FreeBSD userland

After the FreeBSD kernel has finished initializing everything, it mounts the root filesystem so that it can load the first programs from the SD filesystem.

Here are the last messages printed by the kernel:

```
Trying to mount root from
ufs:/dev/mmcsd0s2a [rw,noatime]...
warning: no time-of-day clock regis-
tered, system time will not be set
accurately
```

(In particular, the warning here is expected, since the BBB does not have a battery-backed RTC.)

If you've used FreeBSD or Linux or any similar system before, the remaining boot steps should be quite familiar: The rc system runs a bunch of scripts to set up various standard systems, including network services such as SSHd and NTPd. The very first time you boot, this can take a little while, since some of these services need to set up their initial configurations. Most obviously, the SSHd service needs to create encryption keys for this particular machine.

Finally, the system is ready to accept logins.

```
Wed Sep  4 00:46:40 UTC 2013
FreeBSD/arm (beaglebone) (ttyu0)
login:
```

Most BBB images are set up to automatically configure the Ethernet port and start sshd.

So you should be able to connect remotely using SSH at this point as well.

Using FreeBSD on the BeagleBone Black

The BBB runs a completely standard FreeBSD system, so if you're comfortable with FreeBSD on i386 or amd64, then you should feel right at home.

Here are a few notes to help get you started:

Ethernet: The network interface is "cpsw0". You can configure it with the ifconfig command or edit /etc/rc.conf to set it up on every boot. Most FreeBSD images should have DHCP enabled by default.

Time: Since the BBB does not have a battery-backed clock, you'll need to either set the time manually on boot-up or use NTP to set the time from the network.

Disk: The external micro-SD interface is called "mmcsd0". The standard FreeBSD images for BBB are formatted with two partitions:

* mmcsd0s1 is the FAT slice with U-Boot and other boot files
* mmcsd0s2 is the slice used by FreeBSD

The root partition on mmcsd0s2a is generally formatted with Soft Updates + Journaling (SU+J). SU+J allows the system to reboot quickly when power is removed and reapplied.

eMMC: The 2GB built-in eMMC chip is available as "mmcsd1". By supporting 8-bit transfers, it is significantly faster than the micro-SD interface. The BBB ships with a Linux distribution installed on the eMMC, but you can easily reformat this and use it as an extra drive for FreeBSD. Soon, we expect to be able to install FreeBSD and boot it directly from eMMC.

SU+J: The BBB doesn't have an Off button; you usually just remove power. This does lead to data loss if you have software running when you disconnect power. Using "UFS Soft Updates with Journaling" (UFS SU+J) does not prevent data from being lost, but does seem to do a good job of avoiding fatal filesystem corruption.

Swap: Although 512MB RAM is sufficient for many purposes, you will probably want to enable some swap. For a number of reasons, people are generally using a swap file on the root partition rather than a separate swap partition.

You can use "swapctl -l" to find out if the image you are using already has swap configured. If not, it's easy to add a swap file:

1) Create the file: dd if=/dev/zero of=/usr/swap0 bs=1m count=768

2) Add the following line to /etc/fstab and reboot:

```
md none swap sw,file=/usr/swap0 0 0
```

Ports: If you have network access, then installing a ports tree is quite simple:

```
$ portsnap fetch
$ portsnap extract
```

You can then build and install ports as usual. For example, to install the Apache web server:

```
$ cd /usr/ports/www/apache24
$ make
$ make install
```

Packages: The FreeBSD package team does plan to provide ARM packages compatible with the new package-management tool 'pkg'. As of September 2013 this hasn't yet been implemented.

A number of individuals have had good success using Poudriere to automatically build their own package sets.

USB: USB generally works well on BBB. USB drives, USB network adapters, and printers have all been used successfully. There is one caveat, though: You should not plug any USB peripherals into the BBB unless the BBB is connected to a separate power supply. If you are powering the BBB from a mini-USB cable and try to connect any USB device, the BBB will most likely shut off.

Updating FreeBSD

Once you have FreeBSD up and running, you can download the FreeBSD source code and rebuild directly on the BBB.

Caveats:

* A full system rebuild on the BBB can take as much as two days, depending on a number of factors.

* A full source checkout is over 2G, so won't fit on the eMMC.

* FreeBSD-CURRENT (also called the 'head' branch) is the current development branch; it has the newest features and the newest bugs.

You can use the 'svnlite' command (which is a standard part of FreeBSD now) to check out the source code from the FreeBSD project's Subversion repository:

$ svnlite co http://svn.freebsd.org/base/head /usr/src

$ cd /usr/src

Read /usr/src/UPDATING, especially the summary information near the end that outlines common upgrade scenarios. Generally, a full upgrade from source looks like the following:

```
$ cd /usr/src
$ make buildworld
$ make kernel
<reboot>
$ cd /usr/src
$ mergemaster -p
$ make installworld
$ mergemaster
<reboot>
```

The UPDATING file also explains how to do partial updates, kernel-only updates, and some techniques for doing partial upgrades.

Building Your Own FreeBSD Image

If you are comfortable with the process for building and upgrading FreeBSD from source code, you can use the Crochet tool to build a custom BBB image on a fast i386 or amd64 machine.

In particular, this makes it easy to track the most recent changes to FreeBSD as the support for BBB continues to improve.

Detailed instructions are at:
https://github.com/kientzle/crochet-freebsd;
the following is a quick summary:

1) Get Crochet. You'll need the devel/git package installed, and then you can get a copy of the Crochet scripts:

$ git clone https://github.com/kientzle/crochet-freebsd

To update, use the "git pull" command from inside the source directory.

2) Create a configuration file beagleblack.sh with the following contents:

```
board_setup BeagleBone
option ImageSize 3900mb
option UsrSrc
option UsrPorts
FREEBSD_SRC=${TOPDIR}/src
```

The 'option' lines here preinstall a full FreeBSD source tree in /usr/src and a full ports tree in /usr/ports. Omitting those lines will result in a smaller image.

3) Build the image:

```
$ sudo ./crochet.sh -c beagleblack.sh
```

The script first checks whether you have all the necessary source code and tools. If any are missing, it will print instructions for obtaining them. Once it has all the pieces, a fast PC can compile a complete FreeBSD system and assemble the image in about an hour. ●

Tim Kientzle has been a FreeBSD committer for 10 years and a FreeBSD user for much longer than that. Most recently, he's been working on image-building tools and boot support for BeagleBone and Raspberry Pi.

THE FUTURE OF STORAGE

BY ALLAN JUDE

THE Z FILE SYSTEM (ZFS)

created by Jeff Bonwick and Matthew Ahrens at Sun Microsystems is fundamentally different from previous file systems.

THE KEY difference is that ZFS is, in fact, more than just a file system, as it combines the roles of RAID controller, Volume Manager, and File System.

Most previous file systems were designed to be used on a single device. To overcome this, RAID controllers and volume managers would combine a number of disks into a single logical volume that would then be presented to the file system. A good deal of the power of ZFS comes from the fact that the file system is intimately aware of the physical layout of the underlying storage devices and, as such, is able to make more informed decisions about how to reliably store data and manage I/O operations.

Originally released as part of OpenSolaris, when Sun Microsystems was later acquired by Oracle, it was decided that continued development of ZFS would happen under a closed license. This left the community with ZFS v28 under the original Common Development and Distribution License (CDDL). In order to continue developing and improving this open source fork of ZFS, the OpenZFS project was created—a joint effort between FreeBSD, IllumOS, the ZFS-On-Linux project, and many other developers and vendors. This new OpenZFS (included in FreeBSD 8.4 and 9.2 or later) changed the version number to "v5000 - Feature Flags", to avoid confusion with the continued proprietary development of ZFS at Oracle (currently at v34), and to ensure compatibility and clarity between the various open source versions of ZFS. Rather than continuing to increment the version number, OpenZFS has switched to "Feature Flags" as new features are added. The pools are marked with a property, `feature@featurename`, so that only compatible versions of ZFS will import the pool. Some of these newer properties are read-only backwards compatible, meaning that an older implementation can import the pool and read, but not write to it, because they lack support for the newer features.

What Makes ZFS Different?

The most important feature sets in ZFS are those designed to ensure the integrity of your data. ZFS is a copy-on-write (COW) file system, which means that data is never overwritten in place, but rather the changed blocks are written to a new location on the disk and then the metadata is updated to point to that new location. This ensures that in the case of a shorn write (where a block was being written and was interrupted before it could finish) the original version of the data is not lost or corrupted, as it would be in a traditional file system. In the case of a power failure or system crash, the file is left in an inconsistent state in which it contains a mix of new and old data. Copy-on-write also enables another powerful feature—snapshots. ZFS allows you to instantly create a consistent point-in-time snapshot of a dataset (and optionally of all its child datasets). The new snapshot takes no additional space (aside from a miniscule amount of metadata) and is read-only. Later, when a block is changed, the older block becomes part of the snapshot, rather than being reclaimed as free space. There are now two distinct versions of the file system, the snapshot (what the file system looked like at the time the snapshot was taken) and the live file system (what it looks like now). The only additional space consumed are those blocks that have been changed; the unchanged blocks are shared between the snapshot and the live file system until they are modified. These snapshots can be mounted to recover the older versions of the files that they contain, or the live file system can be rolled back to the time of the snapshot, discarding all modifications since the snapshot was taken. Snapshots are read-only, but they can be used to create a clone of a file system. A clone is a new live file system that contains all the data from its parent while consuming no additional space until it is written to.

These features protect your data from the usual problems: crashes, power failures, accidental deletion/overwriting, etc. However, what about the cases where the problem is less obvious? Disks can suffer from silent corruption, flipped bits, bad cables, and malfunctioning controllers. To solve these problems, ZFS calculates a checksum for every block it writes and stores that along with the metadata. When a block is read, the checksum is again calculated and then compared to the stored checksum; if the two values do not match, something has gone wrong. A traditional file system would have no way of knowing there was a problem, and would happily return the corrupted data. ZFS, on the other hand, will attempt to recover the data from the various forms of redundancy supported by ZFS. When an error is encountered, ZFS increments the relevant counters displayed by the zpool status command. If redundancy is available, ZFS will attempt to correct the problem and continue normally; otherwise, it will return an error instead of corrupted data. The checksum algorithm defaults to fletcher, but the SHA256 cryptographic hashing algorithm is also available, offering a much smaller chance of a hash collision in exchange for a performance penalty.

Future-proof Storage

ZFS is designed to overcome the arbitrary limits placed on previous file systems. For example, the maximum size of a single file on an EXT3 file system is 2^31 (2 TiB), while on EXT4 the limit is 2^44 (16 TiB), compared to 2^55 (32 PiB) on UFS2, and 2^64 (16 EiB) on ZFS. EXT3 is limited to 32,000 subdirectories, with EXT4 limited to 64,000, while ZFS can contain up to 2^48 entries (files and subdirectories) in each directory. The limits in ZFS are designed to be so large that they will never be encountered, rather than just being good enough for the next few years.

Owing to the fact that ZFS is both the volume manager and the file system, it is possible to add additional storage devices to a live system and have the new space available on all the existing file systems in that pool immediately. Each top level device in a zpool is called a vdev, which can be a simple disk or a RAID transform, such as a mirror or RAID-Z array. ZFS file systems (called datasets) each have access to the combined free space of the entire pool. As blocks are allocated, the free space available to the pool (and file system) is decreased. This approach avoids the common pitfall with extensive partitioning where free space becomes fragmented across the partitions.

Doing It in Software Is Better?

Best practices dictate that ZFS be given unencumbered access to the raw disk drives, rather than a single logical volume created by a hardware RAID controller. RAID controllers will generally mask errors and attempt to solve them rather than reporting them to ZFS, leaving ZFS unaware that there is a problem. If a hardware RAID controller is used, it is recommended it be set to IT "Target" or JBOD mode, rather than providing RAID functionality. ZFS includes its

own RAID functionality that is superior.

When creating a ZFS Pool (zpool) there are a number of different redundancy levels to choose from. Striping (RAID0, no redundancy), Mirroring (RAID1 or better with n-way mirrors), and RAID-Z. ZFS mirrors work very much the same as traditional RAID1 (except you can place 3 or more drives into a single mirror set for additional redundancy). However, RAID-Z has some important differences compared to the analogous traditional RAID configurations (RAID5/6/50/60). Compared to RAID5, RAID-Z offers better distribution of parity and eliminates the "RAID5 write hole" in which the data and parity information become inconsistent after an unexpected restart. When data is written to a traditional RAID5 array, the parity information is not updated atomically, meaning that the parity must be written separately after the data has been updated. If something (like a power failure) interrupts this process, then the parity data is actually incorrect, and if the drive containing the data fails, the parity will restore incorrect data. ZFS provides 3 levels of RAID-Z (Z1 through Z3) which provide increasing levels of redundancy in exchange for decreasing levels of usable storage. The number of drive failures the array can withstand corresponds to the name, so a RAID-Z2 array can withstand two drives failing concurrently.

If you create multiple vdevs, for example, two separate mirror sets, ZFS will stripe the data across the two mirrors, providing increased performance and IOPS. Creating a zpool of two or more RAID-Z2 vdevs will effectively create a RAID60 array, striping the data across the redundant vdevs.

ZFS also supports the dataset property `copies`, which controls the number of copies of each block that is stored. The default is 1, but by increasing this value, ZFS will store each block multiple times, increasing the likelihood it can be recovered in the event of a failure or data corruption.

WHEN INITIALIZING NEW POOLS and adding a device to an existing pool, ZFS will perform a whole-device TRIM, erasing all blocks on the device to ensure optimum starting performance.

Faster Is Always Better!

In addition to providing very effective data integrity checks, ZFS is also designed with performance in mind. The first layer of performance is provided by the Adaptive Replacement Cache (ARC), which is resident entirely in RAM. Traditional file systems use a Least Recently Used (LRU) cache, which is simply a list of items in the cache sorted by when each object was most recently used. New items are added to the top of the list, and once the cache is full, items from the bottom of the list are evicted to make room for more active objects. An ARC consists of four lists—the Most Recently Used (MRU) and Most Frequently Used (MFU) objects, plus a ghost list for each. These ghost lists track recently evicted objects to prevent them from being added back to the cache. This increases the cache hit ratio by avoiding objects that have a history of only being used occasionally. Another advantage of using both an MRU and MFU is that scanning an entire file system would normally evict all data from an MRU or LRU cache in favor of this freshly accessed content. In the case of ZFS, since there is also an MFU that only tracks the most frequently used objects, the cache of the most commonly accessed blocks remains. The ARC can detect memory pressure (when another application needs memory)
and will free some of the memory reserved for the ARC. On FreeBSD, the ARC defaults to a maximum of all RAM less 1 GB, but can be restricted using the `vfs.zfs.arc_max` loader tunable.

The ARC can optionally be augmented by a Level 2 ARC (L2ARC). This is one or more SSDs that are used as a read cache. When the ARC is full, other commonly used objects are written to the L2ARC, where they can be more quickly read back than from the main storage pool. The rate at which data is added to the cache devices is limited to prevent prematurely wearing out the SSD with too many writes. Writing to the L2ARC is limited by `vfs.zfs.l2arc_write_max`, except for during the "Turbo Warmup Phase"; until the L2ARC is full (the first block has been evicted to make room for something new), the write limit is increased by the value of

`vfs.zfs.l2arc_write_boost`. OpenZFS also features L2ARC compression controlled by the `secondarycachecompress` dataset property. This increases the effective size of the L2ARC by the compression ratio, but also increases read performance as data is read as quickly as possible but then decompressed, resulting in an even higher effective read speed. L2ARC compression only uses the LZ4 algorithm because of its extremely high decompression performance.

Fine-Grained Control

A great deal of the power of ZFS comes from the fact that each dataset has a set of properties that control how it behaves, and are inherited by its children. A common best practice is to set the `atime` property (which tracks the last access time for each file) to `"off"`. This prevents having to write an update to the metadata of a file each time it is accessed. Another powerful feature of ZFS is transparent compression. It can be enabled and tuned per dataset, so one can compress /usr/src and /usr/ports but disable compression for /usr/ports/distfiles. OpenZFS includes a selection of different compression algorithms including: LZJB (modest compression, modest CPU usage), GZIP1-9 (better compression, but more CPU usage, adjustable), ZLE (compresses runs of 0s, useful in specific cases), and LZ4 (added in v5000, greater compression and less CPU usage than LZJB). LZ4 is a new BSD-licensed high-performance, multi-core scalable compression algorithm. In addition to better compression in less time, it also features extremely fast decompression rates. Compared to the default LZJB compression algorithm used by ZFS, LZ4 is 50% faster when compressing compressible data and over three times faster when attempting to compress incompressible data. The performance on incompressible data is a large improvement; this comes from an "early abort" feature. If ZFS detects that the compression savings is less than 12.5%, then compression is aborted and the block is written uncompressed data, but once decompressed, provides a much higher effective throughput. In addition, decompression is approximately 80% faster; on a modern CPU, LZ4 is capable of compression at 500 MB/s and decompression at 1500 MB/s per CPU core. These numbers mean that for some workloads, compression will actually give increased performance—even with the CPU usage penalty—because data can be read from the disks at the same speed as uncompressed data, but then once decompressed, provides a much higher effective throughput. This also means it is now possible to use dataset compression on file systems that are storing databases, without a heavy latency penalty. LZ4 decompression at 1.5 GB/s on 8k blocks means the additional latency is only 5 microseconds, which is an order of magnitude faster than even the fastest SSDs currently available.

ZFS also provides very fast and accurate dataset, user and group space accounting in addition to quotas and space-reservations. This gives the administrator fine grained control over how space is allocated and allows critical file systems to reserve space to ensure other file systems do not take all of the free space.

On top of all of this, ZFS also features a full suite of delegation features. Delegating various administrative functions such as quota control, snapshotting, replication, ACL management, and control over a dataset's ZFS properties can increase security and flexibility and decrease an administrator's workload. Using these features, it is possible to take consistent backups based on snapshots without root privileges. An administrator could also choose to use a separate dataset for each user's home directory, and delegate control over snapshot creation and compression settings to that user.

Replication—Redundancy Beyond the Node

ZFS also features a powerful replication system. Using the zfs send and zfs receive commands it is possible to send a dataset (and optionally its children) to another dataset, another pool, or another system entirely. ZFS replication also supports incremental sends, sending only the blocks that have changed between a pair of snapshots. OpenZFS includes enhancements to this feature that provide an estimate of how much data will need to be sent, as well as feedback while data is being transferred. This is the basis of PCBSD's Life Preserver feature. A planned feature for the future will also allow resumption of interrupted ZFS send/receive operations.

Harnessing the Power of Solid State Drives

In addition to the L2ARC read-cache discussed earlier, ZFS supports optional log devices, also known as ZFS Intent Log (ZIL). Some workloads, especially databases, require an assurance that the data they have written to disk has actually reached "stable storage." These are called synchronous writes, because the system call does not return until the data has been safely written

to the disk. This additional safety traditionally comes at the cost of performance, but with ZFS it doesn't have to. The ZIL accelerates synchronous transactions by using storage devices (such as SSDs) that are faster and have less latency compared to those used for the main pool. When data is being written and the application requests a guarantee that the data has been safely stored, the data is written to the faster ZIL storage, and then later flushed out to the regular disks, greatly reducing the latency of synchronous writes. In the event of a system crash or power loss, when the ZFS file system is mounted again, the incomplete transactions from the ZIL are replayed, ensuring all of the data is safely in place in the main storage pool. Log devices can be mirrored, but RAID-Z is not supported. When specifying multiple log devices, writes will be load balanced across all devices, further increasing performance. The ZIL is only used for synchronous writes, so will not increase the performance of (nor be busied by) asynchronous workloads.

Open ZFS project (open-zfs.org) was created with the expressed goals of raising awareness about open source ZFS, encouraging **open communication** between the various implementations and vendors, and ensuring consistent reliability, functionality, and performance among all distributions of ZFS.

OpenZFS has also gained TRIM support. Solid State Disks (SSDs) work a bit differently than traditional spinning disks. Due to the way that flash cells wear out over time, SSD's Flash Translation Layer (FTL)—which makes the SSD appear to the system like a typical spinning disk—often moves data to different physical locations in order to wear the cells evenly, and to work around worn-out cells. In order to do this effectively, the SSD's FTL needs to know when a block has been freed (the data stored on it can be overwritten). Without information as to which blocks are no longer in use, the SSD must assume that any block that has ever been written is still in use, and this leads to fragmentation and greatly diminished performance.

When initializing new pools and adding a device to an existing pool, ZFS will perform a whole-device TRIM, erasing all blocks on the device to ensure optimum starting performance. If the device is brand new or has previously been erased, setting the `vfs.zfs.vdev.trim_on_init` sysctl to 0 will skip this step. Statistics about TRIM operations are exposed by the `kstat.zfs.misc.zio_trim` sysctl. In order to avoid excessive TRIM operations and increasing wear on the SSD, ZFS queues the TRIM command when a block is freed, but waits (by default) 64 transaction groups before sending the command to the drive. If a block is reused within that time, it is removed from the TRIM list. The L2ARC also supports TRIM, but based on a time limit instead of number of transaction groups.

OpenZFS—Where Is It Going Next?

The recently founded OpenZFS project (open-zfs.org) was created with the expressed goals of raising awareness about open source ZFS, encouraging open communication between the various implementations and vendors, and ensuring consistent reliability, functionality, and performance among all distributions of ZFS. The project also has a number of ideas for future improvements to ZFS, including: resumable send/receive, ZFS channel programs to allow multiple operations to be complete atomically, device removal, unified ashift handling (for 4k sector "advanced format" drives), increase maximum record size from 128KB to 1MB (preferably in a way compatible with Oracle ZFS v32), platform agnostic encryption, and improvements to deduplication. •

Allan Jude is VP of operations at ScaleEngine Inc., a global HTTP and Video Streaming Content Distribution Network, where he makes extensive use of ZFS on FreeBSD. He is also the on-air host of the video podcasts "BSD Now" with Kris Moore, and "TechSNAP" on JupiterBroadcasting.com. Previously he taught FreeBSD and NetBSD at Mohawk College in Hamilton, Canada, and has 12 years of BSD unix sysadmin experience.

The First PORTS REPORT by Thomas Abthorpe

Welcome to the inaugural Ports Report column for the new *FreeBSD Journal*. I am Thomas Abthorpe, the FreeBSD Ports Management Team secretary, aka portmgr-secretary@. It seemed to be a natural fit for me to be asked to do this column, as I take care of most of the correspondence for the team. So what can you look forward to reading in the Ports Report? It will be a summary of recent activity in the Ports infrastructure, news in the world of Ports, tips and tricks, plus whatever bits of trivia I can slip in.

RESOURCES TO BUILD AND TEST YOUR PORTS

Being a porter is one of the easiest ways to contribute to the FreeBSD Project. Maintaining leafs ports usually requires few resources, and can basically be done on a small home-based system. But what happens if you maintain a port that relies on GTK or QT or something else that requires a substantial number of ports to build first? This is where RedPorts.org comes to the rescue. RedPorts is a cluster of machines maintained by portmgr@ for use by the porter community at large. All you have to do is sign up for an account, and familiarize yourself with checking in your personal ports tree for testing. Once a port has been committed for test purposes, it will be passed off to a builder who will assemble all the dependencies leading up to your port being built. At the end of the process, a log is retained on the server, which you can then review for completeness. Do not get frustrated; sometimes a port has to be tweaked a couple of times before it compiles cleanly.

WHAT'S NEW IN THE PORTS TREE?

The nature of the ports tree is that it is forever evolving, growing, and being updated with new software. From time to time, notable changes are inserted into the infrastructure that improve and/or alter how ports are built. In July 2013, pkg_install stopped being built on 10-CURRENT, this was done in anticipation of pkng. In September 2013, Stack Protector support was introduced for amd64 and i386 in 10-CURRENT.

One of the newest additions to the infrastructure is staging, whichg allows a port to be built into staged directory, instead of getting installed into a production environment. Among other functionality, this allows a package to be created and bundled with a non-privileged account. You can read more about it at http://lists.freebsd.org/pipermail/freebsd-ports-announce/2013-.

VALUE-ADDED SERVICES FROM PORTMGR@

Sometimes, a commit to the ports tree, or for that matter, to the src tree, seems so very trivial, but has far-reaching consequences to userland. When called upon, portmgr@ can perform a full run of the ports tree to see how the change may impact how a port will or will not build. There are generally some high profile changes that will take some coordination—for example, changing the default versions of Perl, Ruby, or Python. There are many others, and you can always call upon your portmgr@ for assistance.

DOING YOUR PART TO IMPROVE THE PORTS TREE

One of the strengths of FreeBSD ports is the diversity of our porters, dedicated volunteers who continually test and update the ports to keep them current and secure. As the infrastructure evolves, new ways of manipulating the Makefile get introduced. The tried-and-true method of validating a port is to run portlint(1). This finds syntax errors, whitespace, and a myriad of other nits to clean up in your port. A recent inclusion into the infrastructure gives enhanced warnings about additional changes you can perform on your Makefile. Set an environment variable DEVELOPER=yes or add it to your /etc/make.conf. By doing this, you may get prompted to alter Makefile to use the newest features being supported.

http://fb.me/portmgr — "Like" us
http://twitter.com/freebsd_portmgr — Follow us
http://blogs.freebsdish.org/portmgr/ — Our blog

NEW PORTS COMMITTERS

It is a long-standing joke that if you submit too many PRs, fix too many ports, and contribute on the mailing lists in a helpful manner, you get punished with a commit bit. So in recent months we have punished the following: John Marino (marino@), a contributor to many BSD projects, notably DragonflyBSD, in which he is responsible for DPorts; Rusmir Dusko (nemysis@), who has concurrently been working with both FreeBSD ports and PC-BSD PBIs; David Chisnall (theraven@), who has spent recent years as a src committer, and with his wealth of experience will be instrumental in getting ports working in the upcoming FreeBSD 10 release cycle; and Danilo Gondolfo, a long-time contributor to the ports tree.

TIPS FOR PERSPECTIVE PORTERS

If you are fortunate enough to maintain a port that just builds with little or no manipulation, then you are quite lucky. This is not the case with all ports. You will often need to patch snippets of code to make it run for FreeBSD. One of the most tedious aspects of this task is maintaining the list of patches in the files subfolder of your port. Instead of running the diff manually to generate your patches, run "make makepatch" from your port, which will assemble all the patches for you. Please also remember to share your patches with the developer of your port, as this will ensure ongoing compatibilityc ●

Thomas Abthorpe is a server administrator with over 20 years in the industry. He got his Ports commit bit August 2007, joined the Ports Management Team in March 2010, and was elected to FreeBSD Core Team in July 2012. When he is not busy doing FreeBSD business, he volunteers as an apprentice bicycle mechanic with Bicycles for Humanity.

Events Calendar

BY DRU LAVIGNE

The following BSD-related conferences are scheduled for the first quarter of 2014. More information about these events, as well as local user group meetings, can be found at bsdevents.org.

FOSDEM ● Feb. 1 & 2, 2014
Brussels, Belgium

https://fosdem.org/2014/

FOSDEM 2014 will take place February 1 & 2 at ULB Campus Solbosch. This annual event is free to attend and features a BSD developer room devoted to BSD presentations. The event will also host the BSDA certification exam.

NYCBSDCon ● Feb. 8, 2014
New York City, NY

http://www.nycbsdcon.org

The NYC BSD Users Group (NYCBUG) will once again hold a one day conference in NYC. The location will be at Suspenders Bar and Restaurant at 111 Broadway. The theme of this year's conference is "The BSDs in Production Environments." As with past NYCBSDCons, the cost will ensure the event is accessible and filled with an array of meetings and discussions reflecting the living, breathing world of the BSDs.

SCALE 12x ● Feb. 21–23, 2014
Los Angeles, CA

http://www.socallinuxexpo.org/scale12x

The 12th annual Southern California Linux Expo will take place February 21–23 at the Hilton Los Angeles Airport Hotel. This BSD-friendly event will include presentations on PC-BSD and FreeNAS and there will be a FreeBSD booth in the expo area. The event will also host the BSDA certification exam on February 23. Registration for the conference is $60.

AsiaBSDCon ● Mar. 13–16, 2014
Tokyo, Japan

http://2014.asiabsdcon.org/

AsiaBSDCon 2014 will take place March 13–16 at the Tokyo University of Science. This conference provides tutorials and presentations and publishes the conference proceedings. This event also hosts a FreeBSD Developer Summit and a Vendor Summit. The Japanese version of the BSDA certification exam will launch at this year's event.

svn update by Glen Barber

The FreeBSD 10-RELEASE cycle is in high gear, and with 9.2-RELEASE officially available, 10 is the primary focus of the Release Engineering team.

In addition to bug fixes and stability enhancements, FreeBSD 10-RELEASE will contain a number of exciting new features.

VIRTIO SUPPORT

WITH VIRTUAL MACHINES, the hypervisor would traditionally emulate real, physical devices to provide to the virtual machine. Emulating the raw device was often inefficient, and would often result in an input/output performance penalty within the virtualized environment.

VirtIO is a specification for paravirtualized input/output devices in a virtual machine. The VirtIO module proivides a shared memory transport between the virtual machine and the hypervisor. This shared memory transport is called the "virtqueue."

The VirtIO PCI driver creates an emulated PCI device that is then made available to the virtual machine. The emulated PCI devices use the virtqueue to directly access memory allocated to the device, resulting in a performance gain within the virtualized environment.

VirtIO was originally developed for the Linux KVM, but has since been adapted to other virtual machine hypervisors, such as BHyVe, VirtualBox, and Qemu.

VirtIO support was **added** in revision (Link: r227652).

BHyVe

BHyVe IS THE BSD Hypervisor, developed by Peter Grehan and Neel Natu. The design goal of BHyVe is to offer a lightweight paravirtualization environment on FreeBSD.

BHyVe requires Intel CPUs with VT-x and Extended Page Table (EPT) support. These features are on all Nehalem CPUs and newer, but not available on Atom CPUs.

BHyVe appeared in FreeBSD 10-CURRENT in revision (Link: r245652).

RDRAND

RDRAND is the Intel CPU instruction set used to access the hardware random number generator.

Software random number generators use seeded entropy obtained from various sources. For example, Ethernet interfaces and software interrupts handlers can be used as sources for entropy to seed random number generation.

Hardware random number generators gather their entropy through physical means, such as thermal "noise" within the device. By using such unpredictable physical entropy sources, the hardware random number generator can gather a higher level of randomness, which, in turn, means a greater possibility of truly random numbers. Having the device directly on the CPU reduces the need for an additional hardware device to be added to the system.

The Intel random number generator is available on the "Bull Mountain" CPU, and present on Ivy Bridge and newer.

The rdrand driver was **added** to FreeBSD 10-CURRENT in revision (Link: r240135).

MULTI-PROCESSOR SUPPORT IN PF

SINCE ORIGINALLY being imported from OpenBSD, one of the performance limitations of PF (Packet Filter) was that it could only run bound to a single CPU. This meant that on multi-processor systems, PF could not take advantage of the additional CPUs, which means that PF would not necessarily show any performance gain when run on 2- or 24- core machines.

Work done on FreeBSD 10-CURRENT introduces multi-processor support to PF, which introduces fine-grain locking support. This allows PF to take advantage of multiple CPUs on the system, which significantly improves performance.

Multi-processor support for PF was introduced in revision (Link: r240233).

The pf firewall, originally from OpenBSD, got **upgraded** to support fine-grain locking and better utilization on multi-cpu machines, which allows it to perform significantly faster.

UNMAPPED IO IN DISK DRIVERS

The FreeBSD kernel maps I/O buffers in the kernel page table. On multi-core systems, the mapping must be flushed on all TLBs (translation lookaside buffers) due to this global mapping. When the number of cores on the system increases, there is a performance bottleneck, since during buffer creation and destruction, the

initiating thread must wait for all other cores on the system to execute.

FreeBSD 10 introduces unmapped I/O buffers, which eliminate the need to perform translation lookaside buffer shootdown for buffer creation and destruction, eliminating up to 30% of system time on I/O-intensive workloads.

Unmapped I/O support was initially **introduced in revision** (Link:r248508) for the ahci(4) and md(4) drivers. Support for additional drivers followed in subsequent revisions.

RASPBERRY PI AND BEAGLEBONE SUPPORT

FreeBSD 10 runs on the Raspberry Pi, BeagleBone, and several other embedded platforms. Although the FreeBSD Project does not yet provide official images for these platforms, several sets of tools exist to create images that can be written to compact flash cards.

One of these tools is "Crochet", which can be used to build images for Raspberry Pi, BeagleBone, and a number of other platforms. Crochet can be found here (Link: https://github.com/kientzle/crochet-freebsd).

Raspberry Pi support was **introduced in revision** (Link: r239922).

CLANG AS THE DEFAULT COMPILER

GCC is no longer part of the default base system on most architectures. The FreeBSD Project has switched from GCC to CLANG as the default compiler. This provides FreeBSD with a more modern, actively-developed default compiler.

Although GCC is not built by default, it is still available in the FreeBSD 10 base system.

The **change** to disable GCC by default was concluded with revision (Link: r255348).

As a hobbyist, Glen Barber became heavily involved with the FreeBSD project around 2007. Since then, he has been involved with various functions, and his latest roles have allowed him to focus on systems administration and release engineering in the Project. Glen lives in Pennsylvania, USA.

this month In FreeBSD

BY DRU LAVIGNE

January is a month of new beginnings—a time to reflect on the past and to consider resolutions to implement in the coming year. Using software as a metaphor, it is a time for a new release that implements known bug fixes and possibly introduces new features! The FreeBSD Project often publishes a new release in January, so let's take a look back in time to see which releases the Project was working on then. Where did your FreeBSD experience begin and which at-the-time "cutting edge" features do you now take for granted?

Jan. 2013

FreeBSD 9.0-RELEASE was announced on January 6, 2013. Being a "dot zero" release, this one was chockfull of new features. One of the interesting aspects of this release is that a number of the larger frameworks, which take significant developer time to design, implement, and test, were sponsored by the FreeBSD Foundation, often in collaboration with other organizations.

For example, the Capsicum framework for application sandboxing is the result of collaboration between the University of Cambridge Computer Laboratory, Google, and the FreeBSD Foundation in which FreeBSD became the reference implementation for new research in application security.

The pluggable congestion control framework, along with five new compression control algorithms, is the result of collaboration between the Foundation and the Swinburne University of Technology's Centre for Advanced Internet Architectures. It represents the cutting edge of TCP compression control research, which is needed in this ever-changing world of networking technologies.

The Foundation collaborated with OMCnet Internet Service GmbH and TransIP BV to implement the Highly Available Storage (HAST) framework, which allows for synchronous block-level replication of any storage media over a TCP/IP network. The Foundation also sponsored the porting of userland Dtrace.

For better or worse, this release finally replaced the "interim" sysinstall framework that Jordan Hubbard introduced for 2.0.5-RELEASE in mid-1995.

The FreeBSD Project dedicated this release to the memory of Dennis M. Ritchie, one of the founding fathers of the UNIX operating system and creator of the C programming language.

2008

The first hour of Marshall Kirk McKusick's course on FreeBSD kernel internals, based on his book, *The Design and Implementation of the FreeBSD Operating System*, was recorded and downloaded in 2008. This course has been given at BSD Conferences and technology companies around the world. **http://www.youtube.com/watch?v=nwbqBdghh6E**

Jan. 2009

FreeBSD 7.1-RELEASE was announced on January 5, 2009. Being the second release in the 7.x series, it didn't introduce too many new features. However, some of the changes it did introduce remind us how far computing has moved along since the turn of the century: the ability to boot from USB devices, the ability to boot from GPT, the ability to use the VESA BIOS for DPMS during suspend and resume, and the ability for traceroute(8) to display an AS number.

Jan. 2004

FreeBSD 5.2-RELEASE was announced on January 12,2004. While many of us remember waiting with baited breath for a very long time for the much anticipated 5.0 (SMP release) in 2003, the other releases in this branch averaged every six months as the fledgling SMP support matured. While 5.2-RELEASE contained a number of significant stability and performance improvements over FreeBSD 5.1, it was still advertised as "a New Technology release that might not be suitable for all users." That is a testament to both the cautious, let's-not-break-production-usage philosophy of the Project and the amount of work and testing needed to move a code base from its uniprocessor assumptions to the new SMP world.

Jan. 1994

This was an interesting time for the newly minted FreeBSD Project. Its first 1.0-RELEASE had moved from EPSILON status and had been unleashed to the world on November 1, 1993. Its future was in a state of flux as the USL vs. BSDI lawsuit marched toward the settlement that was finally announced, minus most of the terms of the agreement, on February 6, 1994.

The settlement allowed the Project to continue its work on FreeBSD 1.1-RELEASE, which was announced on May 6, 1994. That announcement includes this text:

The FreeBSD team is very pleased to announce FreeBSD 1.1 Release, our second full distribution of the FreeBSD Operating System.

FreeBSD 1.1 represents a milestone in our free software efforts, both technically and legally. For quite some time, the future of BSD has been somewhat in doubt due to the UCB/USL lawsuit, and all Net/2 derived distributions have rested on uncertain legal ground. With the resolution of the lawsuit, and subsequent clarification and agreements from USL on our distribution terms, we can bring you this distribution without legal ambiguity, and with clear plans for a fully unencumbered future.

...And the rest, as the saying goes, is history.

THE INTERNET NEEDS YOU

GET CERTIFIED AND GET IN THERE!

Go to the next level with BSD CERTIFICATION

Getting the most out of BSD operating systems requires a serious level of knowledge and expertise

NEED AN EDGE?

BSD Certification can make all the difference.

Today's Internet is complex. Companies need individuals with proven skills to work on some of the most advanced systems on the Net. With BSD Certification

YOU'LL HAVE WHAT IT TAKES!

SHOW YOUR STUFF!

Your commitment and dedication to achieving the **BSD ASSOCIATE CERTIFICATION** can bring you to the attention of companies that need your skills.

BSDCERTIFICATION.ORG

Providing psychometrically valid, globally affordable exams in BSD Systems Administration

Asia BSD Conference 2014

Call for Participation

Tokyo University of Science, Tokyo, Japan

13-16 March, 2014

Official Web Site: http://www.asiabsdcon.org

Contact: secretary@asiabsdcon.org

AsiaBSDCon is a technical conference for users and developers on BSD based systems. AsiaBSDCon 2014 is the 4th international BSD conference in the Asian region and will be held in Tokyo, Japan, in March of 2014. This conference is for anyone developing, deploying and using systems based on FreeBSD, NetBSD, OpenBSD, DragonFlyBSD, Darwin, and MacOS X.

Invited Talks

Dr. Marshall Kirk McKusick, "An Overview of Security in the FreeBSD Kernel"

Eric Allman, "Bambi Meets Godzilla: They Elope - Open Source Meets the Commercial World"

Matthew Ahrens, "OpenZFS ensures the continued excellence of ZFS on FreeBSD, Linux, and illumos"

Scott Long, "Modifying the FreeBSD kernel Netflix streaming servers"

Program Committee

Hiroki Sato (FreeBSD Project, Tokyo Institute of Technology)

George V. Neville-Neil (Neville-Neil Consulting)

Ryan McBride (OpenBSD Project)

Hidetoshi Shimokawa (FreeBSD Project)

Hajimu UMEMOTO (FreeBSD Project)

Masao Uebayashi (The NetBSD Foundation)